

零基础学 Kotlin编程

（美）马尔钦·莫斯卡拉 伊戈尔·沃吉达 著
张 博 译

Android Development with Kotlin

清华大学出版社

零基础学 Kotlin 编程

（美）马尔钦·莫斯卡拉　伊戈尔·沃吉达　著
张　博　译

清华大学出版社
北　京

内 容 简 介

本书详细阐述了与 Kotlin 程序设计相关的基本解决方案，主要包括 Kotlin 语言基础知识、函数、类和对象、泛型、扩展函数和属性、委托机制，以及 Marvel Gallery 项目实战等内容。此外，本书还提供了相应的示例、代码，以帮助读者进一步理解相关方案的实现过程。

本书适合作为高等院校计算机及相关专业的教材和教学参考书，也可作为相关开发人员的自学教材和参考手册。

北京市版权局著作权合同登记号 图字：01-2018-1021

图书在版编目（CIP）数据

零基础学 Kotlin 编程/（美）马尔钦 • 莫斯卡拉（Marcin Moskala），（美）伊戈尔 • 沃吉达（Igor Wojda）著；张博译. —北京：清华大学出版社，2018
书名原文：Android Development with Kotlin
ISBN 978-7-302-50267-8

Ⅰ.①零… Ⅱ.①马… ②伊… ③张… Ⅲ. ①JAVA 语言-程序设计 Ⅳ.①TP312.8

中国版本图书馆 CIP 数据核字（2018）第 114711 号

责任编辑：贾小红
封面设计：刘 超
版式设计：楠竹文化
责任校对：马军令
责任印制：宋 林

出版发行：清华大学出版社
网 址：http://www.tup.com.cn，http://www.wqbook.com
地 址：北京清华大学学研大厦 A 座 **邮 编：**100084
社 总 机：010-62770175 **邮 购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：北京鑫海金澳胶印有限公司
经 销：全国新华书店
开 本：185mm×230mm **印 张：**23 **字 数：**457 千字
版 次：2018 年 6 月第 1 版 **印 次：**2018 年 6 月第 1 次印刷
印 数：1～3000
定 价：120.00 元

产品编号：078393-01

译 者 序

Kotlin 是一种新型语言且具有较好的稳定性，并可在所有 Android 设备上运行，同时还解决了 Java 无法处理的许多问题。Kotlin 为 Android 开发平台引入了许多已被证实的编程概念，使得开发过程变得更加轻松，并可生成更具安全性、表现力和简洁的代码。

本书详细阐述了与 Kotlin 程序设计相关的基本解决方案，主要包括 Kotlin 语言基础知识、函数、类和对象、泛型、扩展函数和属性、委托机制，以及 Marvel Gallery 项目实战等内容。此外，本书还提供了相应的示例、代码，以帮助读者进一步理解相关方案的实现过程。

在本书的翻译过程中，除张博之外，周建娟、李秋霞、程晓磊、黄丽臣、于鑫睿、刘祎、张骞、李垚、张颖、张弢、刘君、李强、李伟、李姣姣、沈旻、翟露洋、刘洋、蔡辉、王福会、杨崇珉、刘璋、刘晓雪、张华臻、刘颙、张满婷等人也参与了本书的翻译工作，在此一并表示感谢。

译　者

前　言

当前，Android 应用程序开发已成普遍之势。在过去的几年中，人们见证了各种工具的发展历程；同时，技术的发展也让我们的生活变得更为便捷。尽管如此，Android 应用程序开发的核心元素却从未发生改变，即 Java。Android 平台可适应新版本的 Java，但一般需要等待很长一段时间，直到最新的 Android 设备达到一定的市场规模。此外，Java 应用程序开发也面临着一系列的挑战，其原因在于：Java 是一种古老的语言，存在着某些设计问题；考虑到向后兼容性的限制，这些问题并不能简单地予以解决。

另一方面，Kotlin 是一种新型语言且具有较好的稳定性，并可在所有 Android 设备上运行，同时还解决了 Java 无法处理的许多问题。Kotlin 为 Android 开发平台引入了许多已被证实的编程概念，使得开发过程变得更加轻松，并可生成更具安全性、表现力和简洁的代码。

本书内容具有易于阅读、实用性强等特征，将帮助读者提升、改进 Android 平台上的 Kotlin 开发体验。另外，本书将在 Java 和解决常见问题的新方法上提供许多“快捷方式”和改进方案。在本书的最后，读者将熟悉 Kotlin 的各项功能和相关工具，并具备在 Kotlin 中开发 Android 应用程序的能力。

本书内容

第 1 章讨论 Kotlin 语言及其特性，同时阐述为何要使用这门语言进行开发。随后将引入 Kotlin 平台，并展示 Kotlin 与 Android 之间的适配方式。

第 2 章主要涉及 Kotlin 的模块构建，并展示该语言中的各种结构、数据类型，以及诸多简化特性。

第 3 章介绍函数的定义和调用方式、函数限定符以及函数的定义位置。

第 4 章介绍 Kotlin 语言中与面向对象相关的各种特性，读者将领略不同的类定义，以及与可读性相关的某些改进特性，如属性操作符重载和中缀调用。

第 5 章讨论了 Kotlin 所支持的函数编程。

第 6 章探讨泛型类、接口以及函数等内容，并深入考察 Kotlin 中的泛型机制。

第 7 章介绍如何向现有类中添加新的操作行为，且无须使用继承机制。除此之外，还将考察集合、流处理方面的简单操作方法。

第 8 章将阐述 Kotlin 中类委托模式的简化方式，这也是该语言内建特性之一。同时还

将展示内建属性委托和自定义委托的应用方式。

第 9 章将通过本书所讨论的诸多特性编写一个功能丰富的 Android 应用程序。

准备工作

当测试并使用本书中的代码时，读者只需安装 Android Studio 即可。第 1 章解释了如何启动新项目、对应代码示例的检查方式，以及大部分代码在未安装任何程序的情况下如何进行测试。

适用读者

当阅读本书时，读者应熟悉以下两方面内容：

- 了解 Java 和面向对象的编程概念，包括对象、类、构造方法、接口、方法、getter 方法、setter 方法和泛型类型。否则，读者将很难理解本书中的相关内容。对此，读者可阅读任何一本 Java 语言的入门书籍。
- 读者应尽量了解 Android 平台，从而可深入理解书中所展示的各项示例，以及 Kotlin 所处理的各种问题。当然，本书对此并不做强制性要求。如果读者是一名拥有 6～12 个月编程经验的 Android 开发者，抑或曾编写了少量的 Android 应用程序，那么本书将十分适合你。另一方面，如果读者了解 OOP 概念，但对 Android 平台了解有限，那么仍可阅读本书的大部分内容。

同时，也希望读者具备开阔的头脑，以及对新技术的渴望之心，这对于程序设计学习来说十分有益。如果本书内容在某些方面引起了读者的好奇之心，那么尽可大胆尝试。

本书约定

代码块通过下列方式设置：

```
val capitol = "England" to "London"
println(capitol.first) // Prints: England
println(capitol.second) // Prints: London
```

代码中的重点内容则采用黑体表示：

```
ext.kotlin_version = '1.1.3'
repositories {
    maven { url 'https://maven.google.com' }
    jcenter()
}
```

命令行输入或输出如下所示：

```
sdk install kotlin
```

 图标表示较为重要的说明事项。

 图标则表示提示信息和操作技巧。

读者反馈和客户支持

欢迎读者对本书的建议或意见予以反馈，以进一步了解读者的阅读喜好。反馈意见对于我们来说十分重要，以便改进我们日后的工作。对此，读者可向 feedback@ packtpub.com 发送邮件，并以书名作为邮件标题。若读者针对某项技术具有专家级的见解，抑或计划撰写书籍或完善某部著作的出版工作，则可访问 www.packtpub.com/ authors。

对于本书的读者，我们将对每一名用户提供竭诚的服务。

资源下载

读者可访问 http://www.packtpub.com 并通过个人账户下载示例代码文件。另外，在 http://www.packtpub.com/support 中注册成功后，我们将以电子邮件的方式将相关文件发与读者。

读者可根据下列步骤下载代码文件：

- 通过个人电子邮件地址和密码登录并注册我们的网站。
- 选择 SUPPORT 选项卡。
- 单击 Code Downloads & Errata。
- 在 Search 文本框中输入书名。
- 选择本书对应的代码文件。
- 从下拉菜单中选择本书的购买方式。
- 单击 Code Download。
- 当文件下载完毕后，确保使用下列最新版本软件解压文件夹：
- Windows 系统下的 WinRAR/7-Zip。
- Mac 系统下的 Zipeg/iZip/UnRarX。
- Linux 系统下的 7-Zip/PeaZip。

另外，读者还可访问 GitHub 获取本书的代码包，对应网址为 https://github.com/PacktPublishing/Android-Development-with-Kotlin。此外，读者还可访问 https://github.com/PacktPublishing/以了解丰富的代码和视频资源。

勘误表

尽管我们在最大程度上做到尽善尽美，但错误依然在所难免。如果读者发现谬误之处，无论是文字错误抑或是代码错误，还望不吝赐教。对于其他读者以及本书的再版工作，这将具有十分重要的意义。对此，读者可访问 http://www.packtpub.com/submit-errata，选取对应书籍，单击 ErrataSubmissionForm 超链接，并输入相关问题的详细内容。经确认后，填写内容将被提交至网站，或添加至现有勘误表中（位于该书籍的 Errata 部分）。

另外，读者还可访问 http://www.packtpub.com/books/content/support 查看之前的勘误表。在搜索框中输入书名后，所需信息将显示于 Errata 项中。

版权须知

一直以来，互联网上的版权问题从未间断，Packt 出版社对此类问题异常重视。若读者在互联网上发现本书任意形式的副本，请告知网络地址或网站名称，我们将对此予以处理。

关于盗版问题，读者可发送邮件至 copyright@packtpub.com。

对于作者的爱护，我们表示衷心的感谢，并于日后向读者呈现更为精彩的作品。

问题解答

若读者对本书有任何疑问，均可发送邮件至 questions@packtpub.com，我们将竭诚为您服务。

目　　录

第 1 章　开启 Kotlin 编程之旅

Kotlin 是一门伟大的语言，可简化 Android 的开发流程。本章将讨论 Kotlin 语言中的具体内容，并考察大量的 Kotlin 示例，以编写更好的 Android 应用程序。欢迎来到 Kotlin 的奇妙旅程，这将改变读者编写代码的思考方式，以及常见编程问题的处理方式。

本章主要涉及以下内容：

- Kotlin 语言中的第一步。
- Kotlin 语言中的操作示例。
- 在 Android Studio 中创建 Kotlin 项目。
- 将现有 Java 项目迁移至 Kotlin 中。
- Kotlin 标准库（stdlib）。
- 为何选择 Kotlin 语言。

1.1　Kotlin 语言简介

Kotlin 是一种静态类型的、兼容 Android 的现代程序设计语言，并修复了许多 Java 中的问题，如空指针异常或代码冗余。Kotlin 是受 Swift、Scala、Groovy、C#等启发而形成的一种语言。Kotlin 由 JetBrains 公司推出，集合了两种语言中的编程经验、应用准则（简洁性和高效性）以及数据类型。经过对其他语言进行分析，Kotlin 尽量避免重复其他语言中的错误，并集中了各种语言中最有效的特性。在采用 Kotlin 编写程序时，明显可感觉到这是一种成熟而优秀的编程语言。

Kotlin 通过提高代码质量、安全性以及开发效率，将应用程序设计提升到一个全新的水平。Google 在 2017 年发布了支持 Android 平台的官方 Kotlin 语言，但 Kotlin 的出现已经有一段时间。Kotlin 社区一直处于非常活跃的状态，而 Kotlin 在 Android 平台上的应用也处于持续增长中。可以将 Kotlin 描述为一个安全的、有表现力的、简洁的、通用的、工具友好的语言，且与 Java 和 JavaScript 具有很强的互操作性。相关特性如下所示：

- 安全性。Kotlin 的安全特性体现在可空性（nullability）和不变性方面。Kotlin 是静态类型的语言，所以每个表达式的类型在编译时均为已知。编译器可以验证试图访问的任何属性或方法，或者某个真实存在的特定类实例。作为一种静态类型语言，Java 也具有该特征；但与 Java 不同，Kotlin 类型系统更加严格（安全）。对此，须明确告知编译器：给定的变量是否可以存储空值。这使得程序在编译时即失效，而

不是在运行期内抛出 NullPointerException，如图 1.1 所示。

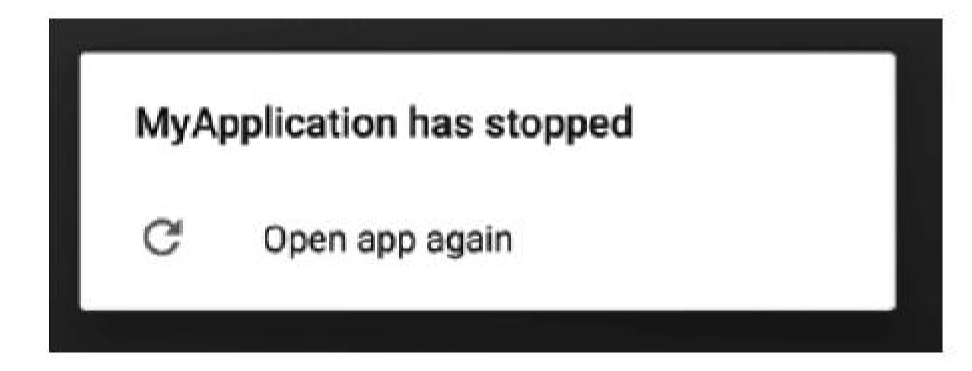


图 1.1

- 更加简单的调试机制。在开发阶段即可快速检测到 bug，而不是在发布后导致应用程序崩溃，从而破坏用户体验。Kotlin 提供了一种使用不可变数据的便捷方法。例如，可以通过提供方便的接口来区分可变（读-写）和不可变（只读）集合（集合中仍然是可变的）。
- 简洁性。Kotlin 消除了 Java 中大多数的冗余内容，仅需更少的代码即可实现常见的任务，因此，即使将 Kotlin 与 Java 8 相比，样板文件代码的数量也大大减少。因此，代码也更容易阅读和理解（表达）。
- 互操作性。Kotlin 被设计成可与 Java（跨语言项目）无缝工作。也就是说，现有的 Java 库和框架可与 Kotlin 协同工作，且不存在任何性能上的损失。许多 Java 库甚至包含特定的 Kotlin 版本，允许在 Kotlin 中实现某些习惯用法。另外。Kotlin 类也可以直接由 Java 代码予以实例化和透明引用，且无须添加任何特殊的语义，反之亦然。因此，可将 Kotlin 合并到现有的 Android 项目中，并实现 Kotlin 和 Java 的混用（若必要的话）。
- 通用性。对此可定位多个平台，包括移动应用（Android）、服务器端应用程序（后端）、桌面应用程序、在浏览器中运行的前端代码，甚至构建系统（Gradle）。

任何编程语言只有在相关工具的辅助下方可表现得更为优异。为此，多种 IDE 均可完美支持 Kotlin，例如 Android Studio、IntelliJ Idea 和 Eclipse，一些常见任务（如代码辅助或重构）同样得到了较好的处理。在发布的每个版本中，Kotlin 团队均致力于提供优秀的 Kotlin 插件。另外，大多数 bug 均可实现快速修复；同时，社区中提出的诸多特性都得以最终实现。



读者可访问 https://youtrack.jetbrains.com/issues/KT 下载 Kotlin 错误跟踪器。
读者可访问 http:// slack. kotlinlang. org/以了解 Kotlin Slack Channel。

当采用 Kotlin 语言时，Android 应用程序开发变得更加高效和轻松。Kotlin 兼容于 JDK 6，因此在 Kotlin 中创建的应用程序甚至可以安全地运行在 Android 设备上（Android 4 之前）。

Kotlin 完美结合了过程式和函数式编程中的概念和元素，并遵循了 *Effective Java，2nd Edition* 一书中的设计理念。该书由 Joshua Bloch 编写，同时也是每一位 Java 开发人员的必备读物。

1.2　示　　例

对于 Android 开发人员来讲，Kotlin 易于学习，其语法与 Java 较为类似；Kotlin 可视为 Java 的自然进化结果。在开始阶段，Kotlin 代码往往无法摆脱 Java 风格；但经过一段时间后，一般会较为自然地转向 Kotlin 方案。下面考察一些较为有趣的 Kotlin 示例。其中，对于常见问题求解，Kotlin 可提供更加简洁、灵活的方法。相关示例尽量保持简单且具有自解释功能，但会涉及本书中的后续内容，读者如不能理解其中内容实属正常现象。本节主要讨论 Kotlin 语言可实现的各种功能，但暂时不关注其中的细节内容，下面首先从变量的声明开始，如下所示：

```
var name = "Igor" // Inferred type is String
name = "Marcin"
```

注意，Kotlin 可以不使用分号。当然，作为可选项，读者依然可继续使用。另外，也不需要指定一个变量类型，因为具体类型是从上下文中推断出来的。每次编译器都可以从上下文找出类型，因而不必显式地予以指定。Kotlin 是一种强类型语言，所以每个变量都应包含适当的类型：

```
var name = "Igor"
name = 2 // Error, because name type is String
```

该变量定义了一个推断 String 类型，因此分配一个不同的值（整数）将导致编译错误。下面是 Kotlin 的改进方案，即如何通过使用字符串模板添加多个字符串：

```
val name = "Marcin"
println("My name is $name") // Prints: My name is Marcin
```

此处并不需要使用+字符连接字符串。在 Kotlin 中，可以很方便地将单个变量，甚至整个表达式合并到字符串常量中，如下所示：

```
val name = "Igor"
    println("My name is ${name.toUpperCase()}")
    // Prints: My name is IGOR
```

在 Java 中，任何变量都可以存储 null 值。在 Kotlin 严格的空保护机制中，则强制要求我们明确标记每个可以存储可空值的变量，如下所示：

```
var a: String = "abc"
a = null // compilation error

var b: String? = "abc"
b = null // It is correct
```

此处将一个问号添加到数据类型中（String 和 String?），即表明变量是可空的（可以存储空引用）；否则将无法为其分配一个空引用。Kotlin 还允许以适当的方式处理可空变量，例如可使用安全调用操作符调用基于空变量的方法，如下所示：

```
savedInstanceState?.doSomething
```

仅当 savedInstanceState 包含非空值的情况下才会调用 doSomething 方法；否则该方法调用将被忽略。这是 Kotlin 避免 Java 中常见的空指针异常的一种安全方法。

Kotlin 还涵盖了一些新的数据类型，例如 Range 数据类型，可定义相应的终止范围，如下所示：

```
for (i in 1..10) {
   print(i)
} // 12345678910
```

另外，Kotlin 还引入了 Pair 数据类型，当与中缀标记法结合使用时，可装载常见的数值对，如下所示：

```
val capitol = "England" to "London"
println(capitol.first) // Prints: England
println(capitol.second) // Prints: London
```

对此，可采用解构声明将其分解为独立变量，如下所示：

```
val (country, city) = capitol
println(country) // Prints: England
println(city) // Prints: London
```

甚至还可遍历数值对列表，如下所示：

```
val capitols = listOf("England" to "London", "Poland" to "Warsaw")
for ((country, city) in capitols) {
    println("Capitol of $country is $city")
}

// Prints:
// Capitol of England is London
// Capitol of Poland is Warsaw
```

除此之外，还可使用 forEach 函数，如下所示：

```
val capitols = listOf("England" to "London", "Poland" to "Warsaw")
capitols.forEach { (country, city) ->
    println("Capitol of $country is $city")
}
```

需要注意的是，Kotlin 提供了一组接口和帮助方法（List 和 MutableList，Set 和 MutableSet，Map 和 MutableMap 等），进而区分可变和只读集合，如下所示：

```
val list = listOf(1, 2, 3, 4, 5, 6) // Inferred type is List
val mutableList = mutableListOf(1, 2, 3, 4, 5, 6)
// Inferred type is MutableList
```

这里，只读集合表示，初始化之后集合状态将无法变更（无法添加、移除数据项）；相比之下，可变集合则可对状态进行调整。

当采用 Lambda 表达式时，还可通过非常简洁的方式使用 Android 框架，如下所示：

```
view.setOnClickListener {
    println("Click")
}
```

Kotlin 标准库中定义了多种函数，并以简洁的方式执行各项集合操作。例如，可在列表上进行流处理，如下所示：

```
val text = capitols.map { (country,  ) -> country.toUpperCase() }
                   .onEach { println(it) }
                   .filter { it.startsWith("P") }
                   .joinToString (prefix = "Countries prefix P:")
// Prints: ENGLAND POLAND
println(text) // Prints: Countries prefix P: POLAND
.joinToString (prefix = "Countries prefix P:")
```

注意，无须向某个 Lambda 传递参数，可自定义 Lambda，并以一种全新的方式编写代码。Lambda 仅在 Android Marshmallow（或更新的版本）中运行特定的代码片段，如下所示：

```
inline fun supportsMarshmallow(code: () -> Unit) {
    if(Build.VERSION.SDK INT >= Build.VERSION CODES.M)
    code()
}

//usage
supportsMarshmallow {
    println("This code will only run on Android Nougat and newer")
}
```

可以使用 doAsync 函数轻松地在主线程上进行异步请求并显示响应结果，如下所示：

```
doAsync {
    var result = runLongTask() // runs on background thread

    uiThread {
        toast(result)          // run on main thread
    }
}
```

另外，智能转换（Smart Cast）可采用更加简洁的方式编写代码，如下所示：

```
if (x is String) {
    print(x.length) // x is automatically casted to String
}

x.length //error, x is not casted to a String outside if block

if (x !is String)
    return

x.length // x is automatically casted to String
```

在检测完毕后，Kotlin 编译器知晓变量 x 表示为 String 类型，因而自动将其转换为 String 类型，进而调用全部方法并访问 String 类中的全部属性，且无须显式地进行转换。

某些时候，一些简单的函数返回单个表达式的值。在这种情况下，可以使用基于表达式体（expression body）的函数来简化语法，如下所示：

```
fun sum(a: Int, b: Int) = a + b
println (sum(2 + 4)) // Prints: 6
```

当采用默认参数语法时，可针对各个函数参数定义默认值，并通过多种方式进行调用，如下所示：

```
fun printMessage(product: String, amount: Int = 0,
    name: String = "Anonymous") {
    println("$name has $amount $product")
}
printMessage("oranges") // Prints: Anonymous has 0 oranges
printMessage("oranges", 10) // Prints: Anonymous has 10 oranges
printMessage("oranges", 10, "Johny")
// Prints: Johny has 10 oranges
```

唯一的限制条件是，若未设置默认值，需要提供全部参数。除此之外，还可使用命名参数语法确定函数参数，如下所示：

```
printMessage("oranges", name = "Bill")
```

当在函数调用过程中调用包含多个参数的函数时，这也提升了代码的可读性。

数据类提供了一种非常简单的方法来定义和操作来自数据模型的类。当定义一个数据类时，可在类名之前使用 data 限定符，如下所示：

```
data class Ball(var size:Int, val color:String)

val ball = Ball(12, "Red")
println(ball) // Prints: Ball(size=12, color=Red)
```

注意，此处生成一个类实例，且包含了具有可读性的字符串表达形式；同时，无须使用 new 关键字初始化当前类。除此之外，还可方便地创建该类的副本，如下所示：

```
val ball = Ball(12, "Red")
println(ball) // prints: Ball(size=12, color=Red)
val smallBall = ball.copy(size = 3)
println(smallBall) // prints: Ball(size=3, color=Red)
smallBall.size++
println(smallBall) // prints: Ball(size=4, color=Red)
println(ball) // prints: Ball(size=12, color=Red)
```

上述构造过程可简单、方便地与不可变对象协同工作。

扩展（Extension）是 Kotlin 语言中的一个优异特性，并以此向现有类中添加新的行为（方法或属性），且无须改变其实现。某些时候，当与某个库或框架协同工作时，须针对特定类定义附加方法或属性，则可通过扩展添加所缺失的成员。扩展降低了代码的冗余性，并移除了 Java 中的各种工具函数（例如 StringUtils 类）。针对自定义类、第三方库，甚至是 Android 框架类，均可方便地定义扩展。首先，ImageView 并不具备从网络中加载图像的能力，因而可添加 loadImage 扩展，并通过 Picasso 库加载图像（Android 的图像加载库），如下所示：

```
fun ImageView.loadUrl(url: String) {
    Picasso.with(context).load(url).into(this)
}

// usage
imageView.loadUrl("www.test.com\\image1.png")
```

除此之外，还可向 Activity 类添加 toast 方法，如下所示：

```
fun Context.toast(text:String) {
    Toast.makeText(this, text, Toast.LENGTH SHORT).show()
}
```

```
// usage (inside Activity class)
toast("Hello")
```

扩展的应用远不止这些，从而可极大地减少代码量。当使用 Kotlin 时，还可使用 Lambda 进一步简化 Kotlin 代码。

Kotlin 的接口可以包含默认实现，只要它们不保留任何状态，如下所示：

```
interface BasicData {
    val email:String
    val name:String
    get() = email.substringBefore("@")
}
```

在 Android 中，存在多种应用场合须延迟对象的初始化操作。对此，可使用委托机制，如下所示：

```
val retrofit by lazy {
    Retrofit.Builder()
        .baseUrl("https://www.github.com")
        .addConverterFactory(MoshiConverterFactory.create())
        .build()
}
```

其中，Retrofit（较为流行的 Android 网络框架）属性初始化操作将被延迟，直至首次访问该值。延迟初始化行为可导致更快的 Android 应用程序启动时间——加载过程将延迟至变量首次访问时。当在某个类中初始化多个对象时，这可视作一种较好的方法，尤其是仅使用部分对象时（针对某些特殊的类应用方案，仅须使用一些特定类）；或者是类创建完毕后，仅须使用部分对象时。

上述内容简单扼要地描述了 Kotlin 可实现的某些任务，下面将讨论 Kotlin 的应用方式。

1.3　处理 Kotlin 代码

Kotlin 代码的管理和运行机制存在多种方式，本节主要考察 Android Studio 和 Kotlin Playground。

1.3.1　Kotlin Playground

读者可通过 Kotlin Playground 感受 Kotlin 代码，且无须安装其他软件，如图 1.2 所示。Kotlin Playground 的对应网址为 https://try. kotlinlang. org。读者可采用 JavaScript 或 JVM 的 Kotlin 实现运行代码，并方便地在 Kotlin 的不同版本之间进行切换。本书的全部代码示例

不存在 Android 框架依赖性，并可通过 Kotlin Playground 予以执行。

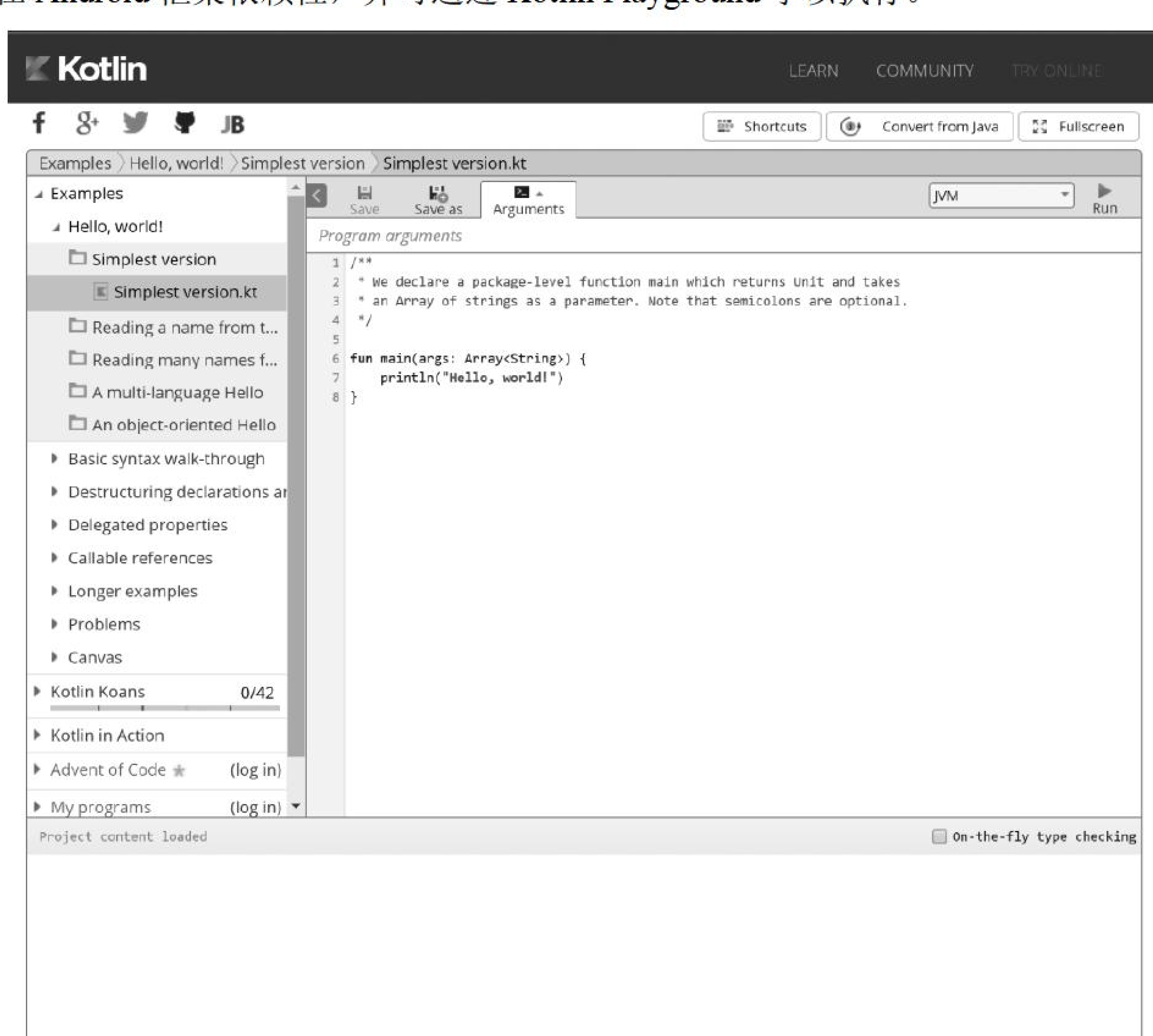


图 1.2

其中，main 函数表示为 Kotlin 应用程序入口点，当应用程序启动时即调用该函数。因此，需要将书中的代码示例置于该函数体中。另外，读者可直接放置代码，或者放置涵盖更多 Kotlin 代码的函数调用，如下所示：

```
fun main(args: Array<String>) {
    println("Hello, world!")
}
```



Android 应用程序包含多个入口点。main 函数被 Android 框架隐式调用，因而无法使用该函数在 Android 平台上运行 Kotlin 代码。

1.3.2　Android Studio

Android Studio 中的现有全部工具均可与 Kotlin 代码协同工作。据此，用户可使用调试工具、Lint 检查、代码提示以及代码重构等内容。大多数内容与 Java 的工作方式并无太多差异，其中较为显著的变化是 Kotlin 语法。全部工作需要在当前项目中配置 Kotlin。

Android 应用程序包含多个入口点（可启动应用程序中不同的组件），同时具有 Android 框架依赖性。当运行本书中的示例代码时，需要扩展 Activity 类，并将代码置于其中。

（1）在项目中配置 Kotlin

自 Android Studio 3.0 之后，即加入了对 Kotlin 语言的工具支持。此处不需要安装 Kotlin 插件，Kotlin 已深度集成至 Android 开发处理中。

当在 Android Studio 2.x 环境下使用 Kotlin 时，须手动安装 Kotlin 插件。在安装过程中，可选择 Android Studio | File | Settings | Plugins | Install JetBrains plugin... | Kotlin，并单击 Install 按钮，如图 1.3 所示。

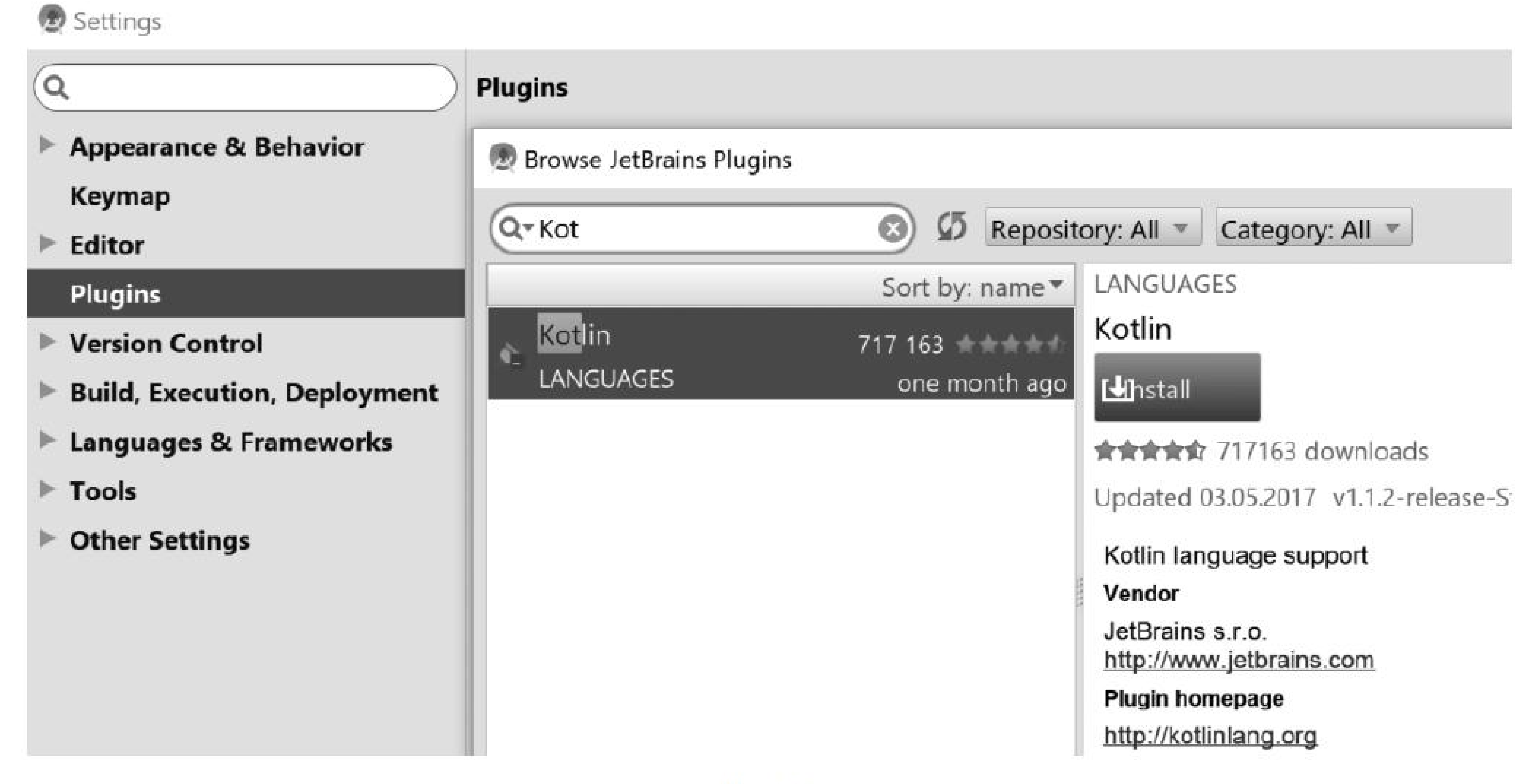


图 1.3

为了正常使用 Kotlin，须在项目中对 Kotlin 进行配置。针对现有 Java 项目，须运行 Configure Kotlin in project（在 Windows 环境下，对应快捷键为 Shift+Ctrl+A；在 macOS 环境下，快捷键为 command+shift+A）；或者使用 Tools | Kotlin | Configure Kotlin in Project 菜单项，如图 1.4 所示。

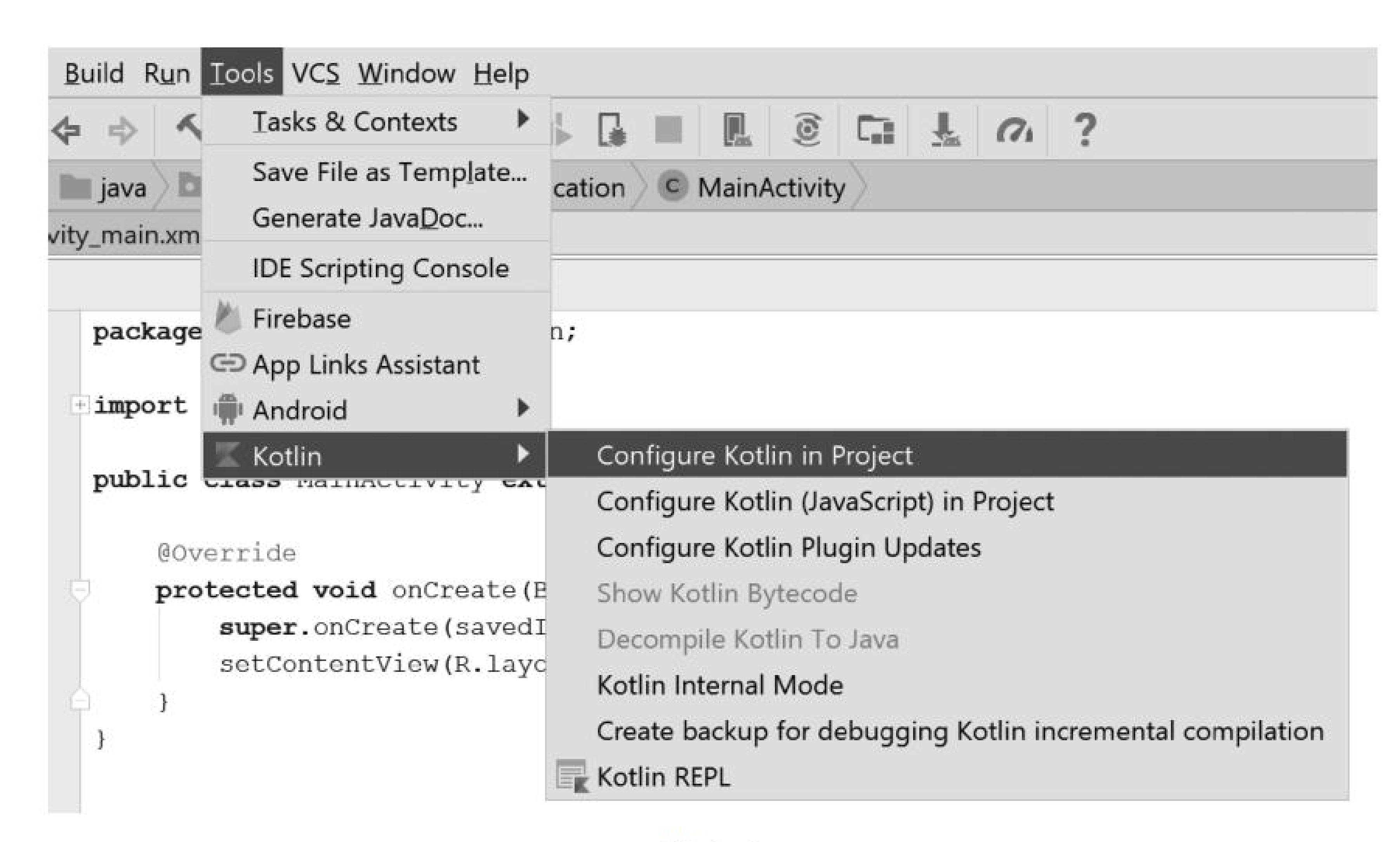


图 1.4

随后可选择 Android with Gradle，如图 1.5 所示。

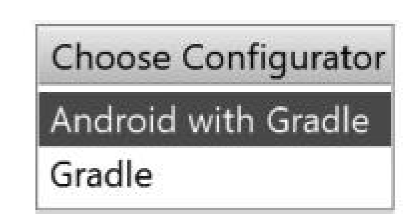


图 1.5

最后需要选取所需模块以及相应的 Kotlin 版本，如图 1.6 所示。

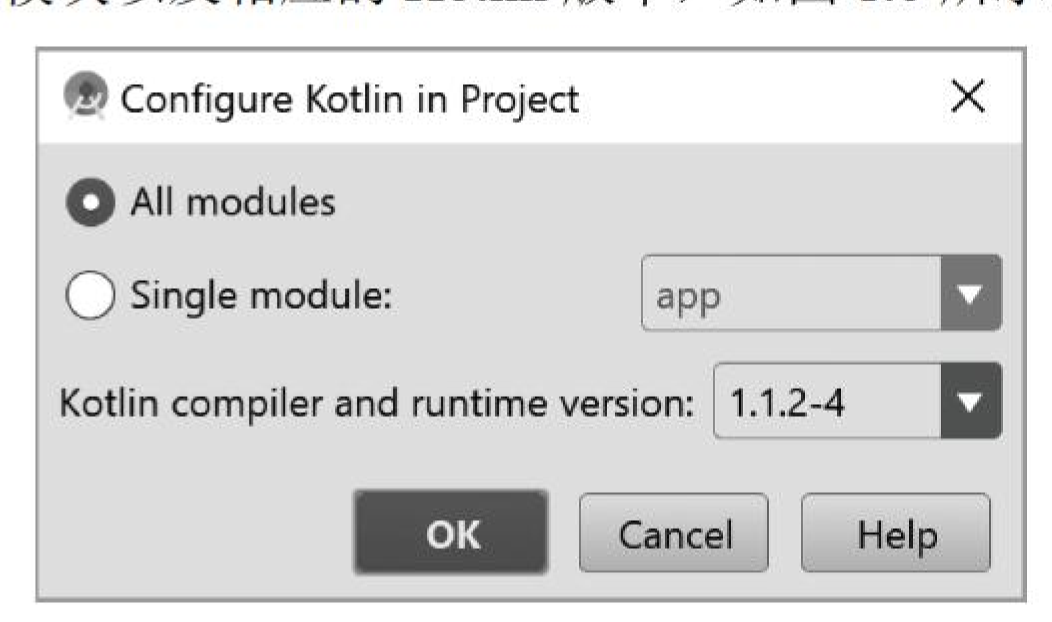


图 1.6

上述配置方案也适用于现有的、Java 最初创建的全部 Android 项目。自 Android Studio 3.0 起，在生成新项目的同时，还可同时选中 Include Kotlin support 复选框，如图 1.7 所示。

图 1.7

在图 1.7 所示的两种方案中，通过添加 Kotlin 依赖关系，Configure Kotlin in project 命令将更新与当前模块对应的 build.gradle 根文件和 build.gradle 文件；除此之外，还将向当前 Android 模块添加 Kotlin 插件。在本书编写时，Android Studio 3 版本尚不支持此类功能，但可从预览版本中查看构建脚本，如下所示：

```
// build.gradle file in project root folder
buildscript {
    ext.kotlin version = '1.1'

    repositories {
        google()
        jcenter()
    }
    dependencies {
       classpath 'com.android.tools.build:gradle:3.0.0-alpha9'
       classpath "org.jetbrains.kotlin:kotlin-gradleplugin:$
              kotlin version"
    }
}

...
// build.gradle file in the selected modules
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
...
dependencies {
    ...
```

```
    implementation 'com.android.support.constraint:constraintlayout:
    1.0.2'
}
...
```



在 Gradle 3.x Android 插件之前（Android Studio 3.0 所提供），一般采用编译（而非实现）依赖性配置。

当更新 Kotlin 版本时，须修改文件 build.gradle（位于项目根目录）中的 kotlin_version 变量值。Gradle 文件中的变化内容意味着当前项目须实现同步化。因此，Gradle 将更新其配置内容，并下载全部所需的依赖关系，如图 1.8 所示。

Gradle files have changed since last project sync. A project sync may be necessary for the IDE to work properly.　　Sync Now

图 1.8

（2）在 Android 新项目中使用 Kotlin

对于 Android Studio 3.x 中创建的最新 Kotlin 项目，主要操作均已在 Kotlin 中加以定义，随后即可开始编写 Kotlin 代码，如图 1.9 所示。

MainActivity.kt

```
package com.test.myapplication

import ...

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

图 1.9

Kotlin 文件的添加方式与 Java 类似，即右击数据包并选择 New | Kotlin File/Class 命令，如图 1.10 所示。



图 1.10



图 1.10 中，IDE 显示 Kotlin File/Class，而非 Kotlin class（类似于 Java class），其原因在于，单一文件中定义了多个成员。第 2 章将对此予以详细解释。

需要注意的是，Kotlin 源文件可置于 java 源文件夹中。对此，可生成一个新的 Kotlin 文件夹，但此处并非必需，如图 1.11 所示。

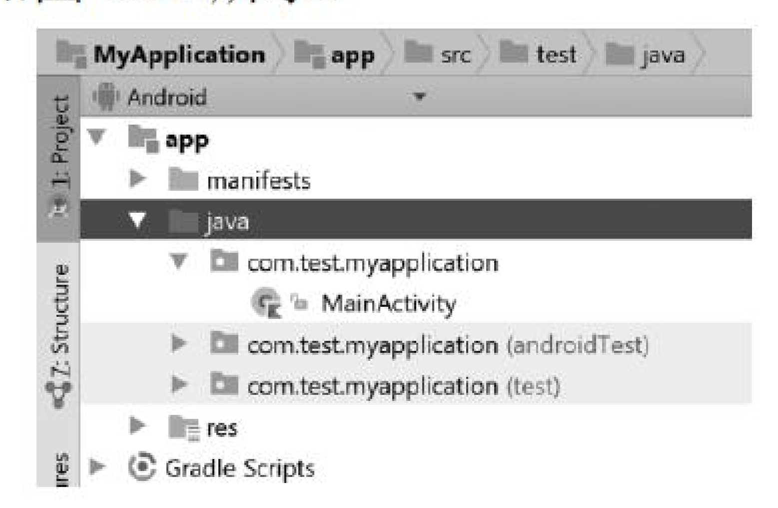


图 1.11

项目的运行和调试与 Java 基本相同，除了在项目中配置 Kotlin 之外，并不需要其他附加步骤，如图 1.12 所示。

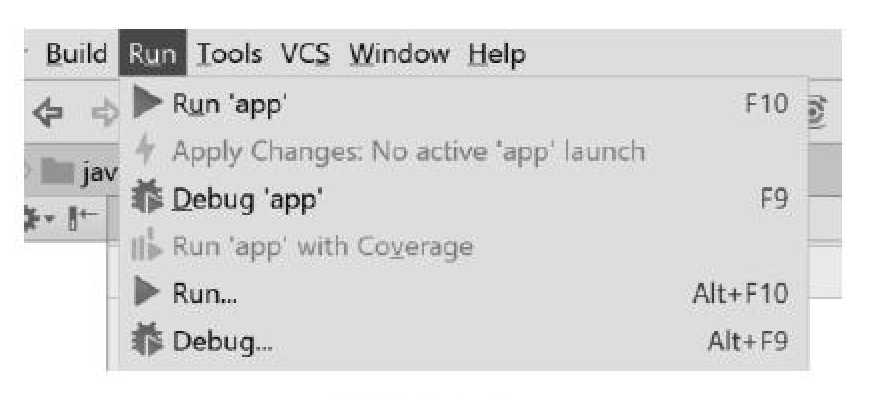


图 1.12

自 Android Studio 3.0 起，通过 Android 模板可选取某种语言，即 Configure Activity 安装向导，如图 1.13 所示。

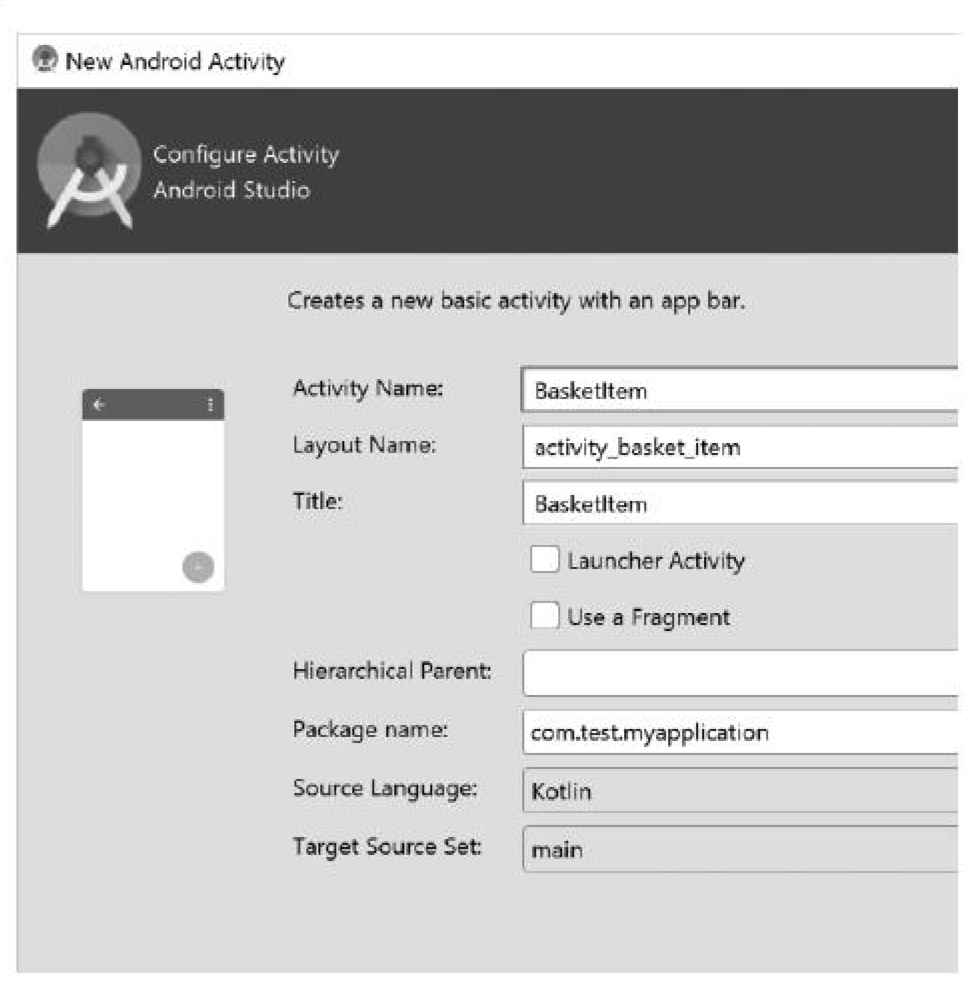


图 1.13

（3）Java-Kotlin 转换器（J2K）

现有 Java 项目的移植操作也较为简单，对此，可在同一项目中共同使用 Java 和 Kotlin。通过 Java to Kotlin converter(J2K)，存在两种方式可将现有的 Java 代码转换为 Kotlin 代码。

第一种方法是使用 convert Java File to Kotlin 命令将全部 Java 文件转换为 Kotlin 文件（在 Windows 环境下，快捷键为 Shift+Alt+Ctrl+K；在 macOS 环境下，快捷键为 option+shift+command+K），该方法工作良好。第二种方法则是将 Java 代码粘贴至现有的 Kotlin 文件中，对应代码也将被转换（此时会出现一个包含转换提示信息的对话框）。

如果读者尚不了解如何在 Kotlin 中编写特定代码，首先可在 Java 中编写代码，并于随后将其复制至 Kotlin 文件中。虽然转换后的代码并不是 Kotlin 的惯用版本，但依然可正常工作。同时，IDE 会显示多种代码转换提示，以改善代码的质量。在转换前，应确保 Java 代码有效；另外，转换工具易受到外界影响，即使缺失一个分号，转换过程也会失败。基于 Java 交互操作的 J2K 转换器可将 Kotlin 以渐进方式引入至现有的项目中（例如一次转换一个类）。

（4）运行 Kotlin 代码的替代方案

Android Studio 提供了运行 Kotlin 代码的替代方案，且无须运行 Android 应用程序。当对某些 Kotlin 代码（独立于较为耗时的 Android 编译和部署处理）进行快速测试时，这一方案十分有用。

Kotlin 代码的运行方式使用了 Kotlin REPL，如图 1.14 所示。其中，REPL 是一种简单的语言 shell 命令，读取单一的用户输入，对其进行评估并输出结果。

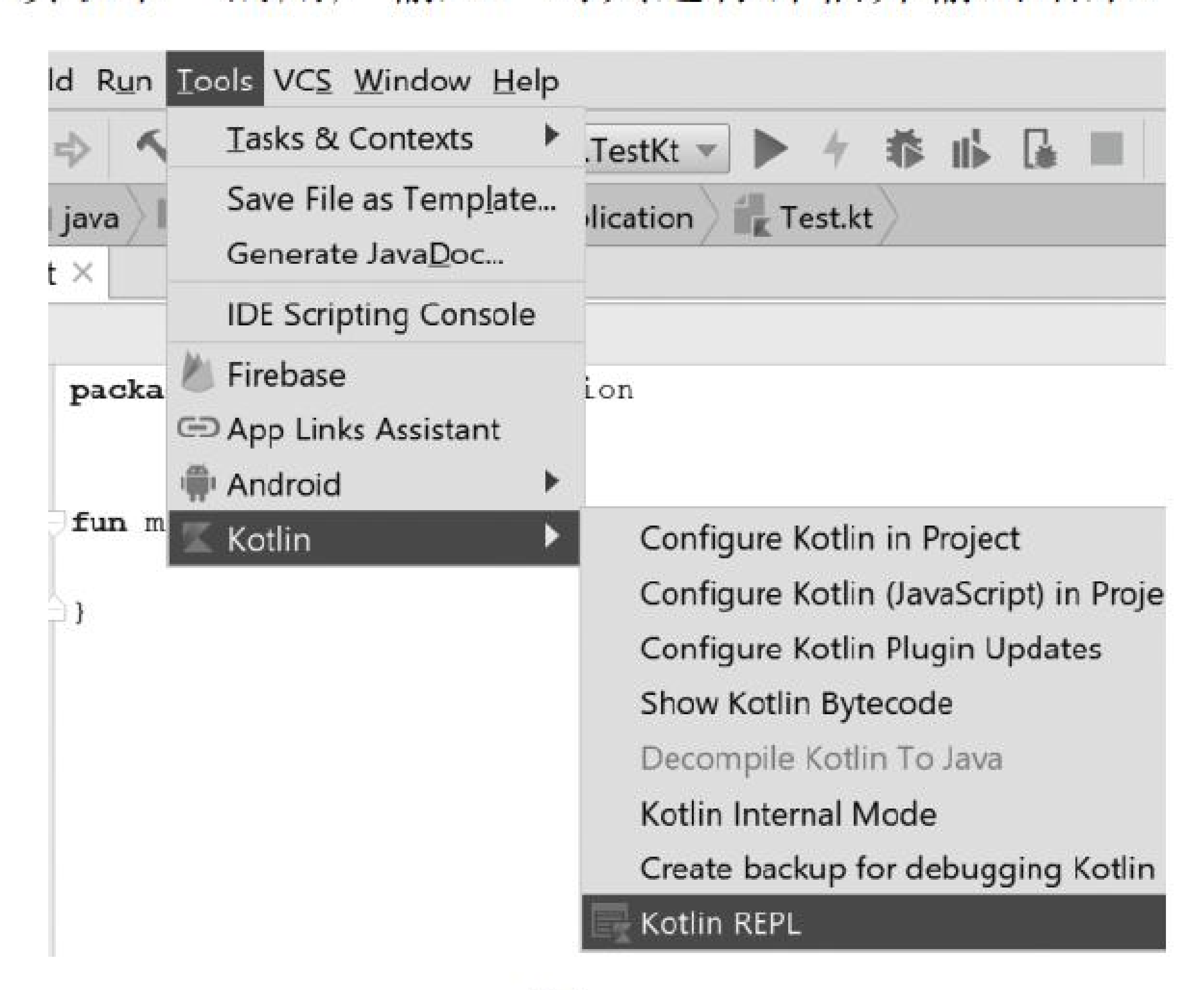


图 1.14

REPL 类似于命令行工具，如图 1.15 所示，但会提供全部所需的代码提示内容，同时还可访问当前项目中定义的各种结构（例如类、接口、上层函数等）。

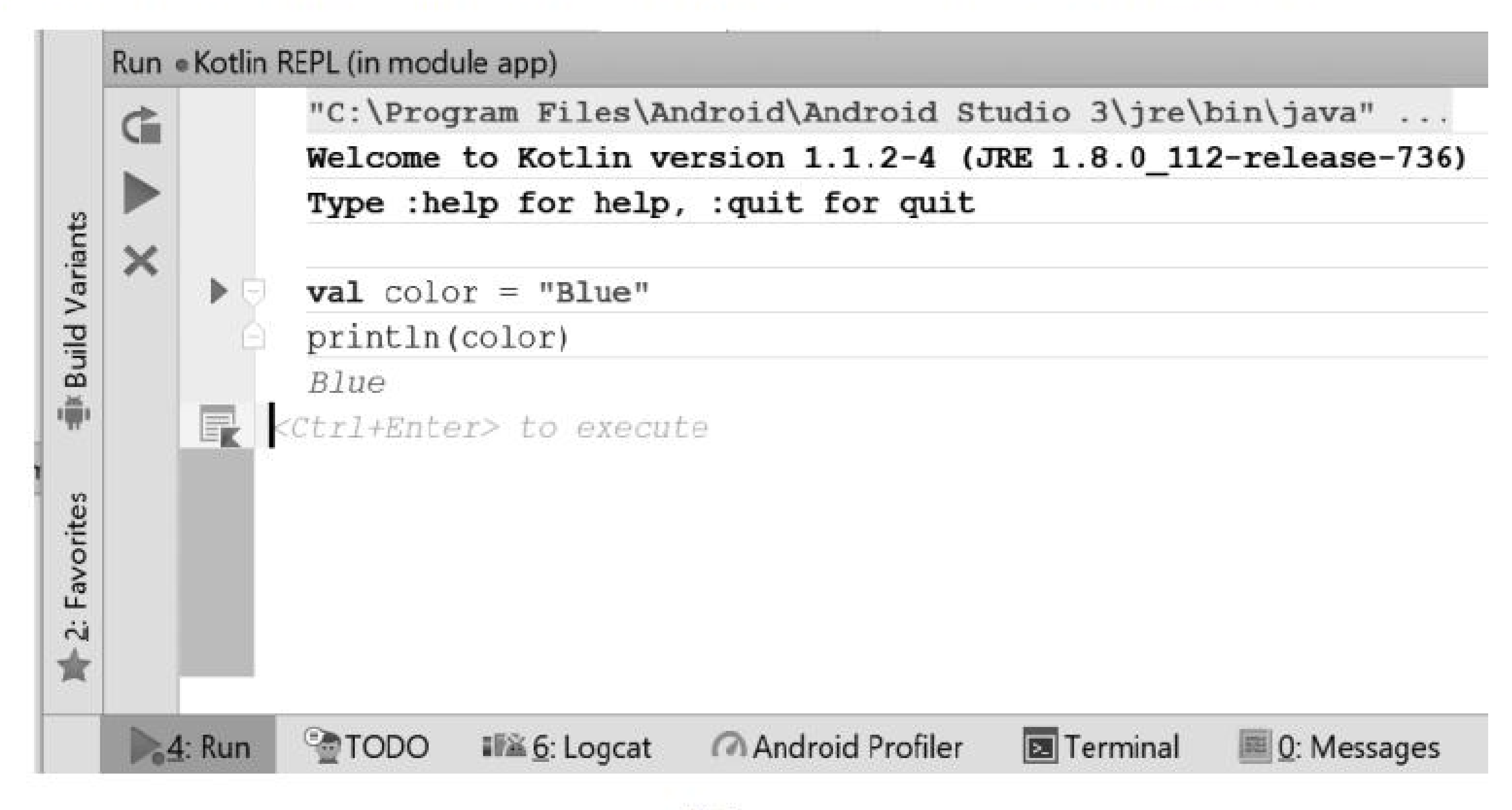


图 1.15

速度是 REPL 的最大优势，因而可快速对 Kotlin 代码进行测试。

1.4　Kotlin 底层机制

虽然本书主要关注 Android 平台，但不应忘记，Kotlin 可编译至多个平台上。相应地，Kotlin 可编译为 Java 字节码，随后则是 Dalvik 字节码。图 1.16 显示了针对 Android 平台的、Kotlin 构建处理过程的简化版本。

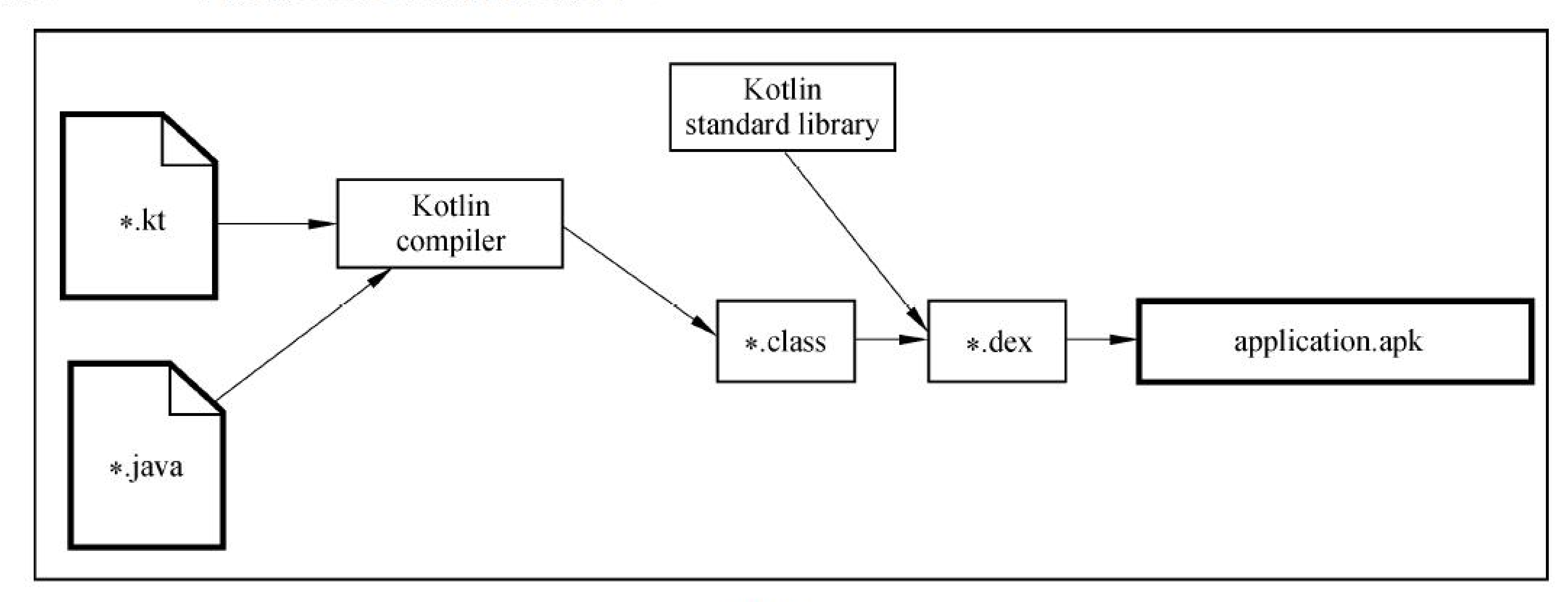


图 1.16

- 包含.java 扩展名的文件包含 Java 代码。
- 包含.kt 扩展名的文件包含 Kotlin 代码。

- 包含.class 扩展名的文件包含 Java 字节码。
- 包含.dex 扩展名的文件包含 Dalvik 字节码。
- 包含.apk 扩展名的文件包含 AndroidManifest 文件、资源文件以及.dex 文件。

对于纯粹的 Kotlin 项目，仅使用到 Kotlin 项目，但 Kotlin 也支持跨语言项目。其中，可在同一 Android 项目中使用 Kotlin 和 Java。此时，两种编译器均会用于编译 Android 应用程序，最终结果将实现类级别的合并。

Kotlin 标准库（stdlib）是一个小型库，并随同 Kotlin 一同发布。该标准库需要运行 Kotlin 编写的应用程序，并在构建处理过程中自动添加至应用程序中。



在 Kotlin 1.1 中，需要配置 kotlin-runtime 以运行 Kotlin 编写的应用程序。实际上，Kotlin 1.1 包含了两种缺陷（kotlin-runtime 和 kotlin-stdlib），并涉及多个 Kotlin 数据包。为了减少混淆，kotlin-runtime 和 kotlin-stdlib 在 Kotlin 1.2 版本中将合并至单一的 kotlin-stdlib 中。自 Kotlin 1.2 起，需要使用到 kotlin-stdlib 以运行 Kotlin 编写的应用程序。

Kotlin 标准库提供了 Kotlin 各项工作所需的必备元素，其中包括：

- 数组、集合、表、范围等数据类型。
- 扩展。
- 高阶函数。
- 与字符串和字符序列协同工作的各种工具方法。
- JDK 类扩展，进而可方便地与文件、IO 以及线程协同工作。

1.5　Kotlin 的其他优势

Kotlin 针对 JetBrains 提供了强有力的商业支持，并对众多流行的编程语言配置了 IDE（Android Studio 基于 JetBrains IntelliJ IDEA）。JetBrains 计划改进代码的质量以及团队的开发效率，因而须通过一种语言解决 Java 中的种种问题，同时提供与 Java 之间的无缝交互操作。对此，并无相关语言可满足此类条件，JetBrains 最终决定设计自己的语言，并启动了 Kotlin 项目。目前，Kotlin 应用于其旗舰产品中。其中，一些开发人员结合 Java 进行 Kotlin 开发；而另一些开发者则直接使用 Kotlin 产品。

Kotlin 是一门十分成熟的语言，实际上，该语言于多年之前即已出现，随后 Google 通过官方宣布，Android 对 Kotlin 予以支持（2010 年 11 月 8 日），如图 1.17 所示。

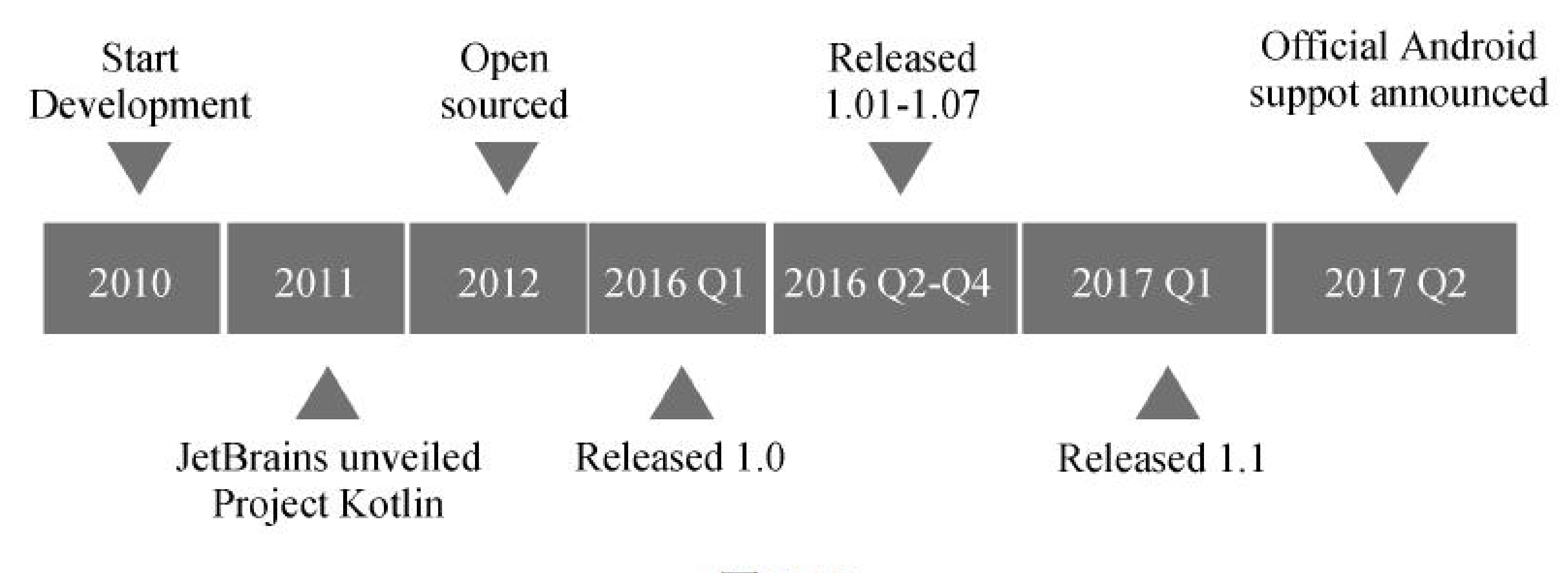


图 1.17



Jet 则是 Kotlin 语言的最初名称。后来，JetBrains 团队确定将其更名为 Kotlin。该名称源自圣彼得堡附近的科特林岛，类似于 Java 取名于印度尼西亚的爪哇岛。

Kotlin 1.0 版本于 2016 年发布，随后，越来越多的公司开始支持 Kotlin 项目。其中，Gradle 添加了对 Kotlin 的脚本支持；作为一家最大的 Android 库开发商，Square 也对 Kotlin 提供了大力支持；最终，Google 宣布在 Android 平台上对 Kotlin 予以支持。这也意味着，Android 发布的每一项工具均兼容于 Java 和 Kotlin。Google 和 JetBrains 最终达成合作关系，并创建了 Kotlin 非营利组织，负责该语言的维护和开发。所有努力旨在推动 Kotlin 语言的市场份额。

Kotlin 类似于 Apple 公司推出的 Swift 编程语言。实际上，二者间具有很多相似之处，许多文章着重分析了两种语言之间的差异性（而非相似性）。对于渴望学习 Android 和 iOS 应用程序开发的人士来讲，Kotlin 语言是一种较好的选择方案。另有计划表明，Kotlin 以后可导入至 iOS（Kotlin/Native）中，或许学习 Swift 语言将不再必需。另外，Kotlin 还可实现全栈开发，因而可开发服务器端应用程序，以及前端客户端应用程序，并在移动客户端上共享同一数据模型。

1.6 本 章 小 结

本章讨论了 Kotlin 语言与 Android 开发之间的关系，以及 Kotlin 与现有项目之间的整合方式。另外，本章还考察了相关示例，并展示了 Kotlin 语言的简洁性和安全性。

第 2 章主要学习 Kotlin 语言的构造模块，以及使用 Kotlin 语言开发 Android 应用程序时须具备的基础知识。

第 2 章　Kotlin 语言基础知识

本章主要讨论基本构建模块，这也是 Kotlin 语言中的核心元素。每一个模块自身并不具备太多含义，经适当组合后，即可创建功能强大的语言结构。本章将探讨 Kotlin 语言的类型机制，并介绍严格的空保护以及智能转换机制。除此之外，还将考察 JVM 中的一些新增操作符。与 Java 语言相比，Kotlin 提供了多种改进措施。同时，本章还将阐述最新的应用程序控制流方法，并采用统一方式处理等式问题。

本章主要涉及以下内容：

- 变量、值和常量。
- 类型推断。
- 严格的空安全机制。
- 智能转换。
- Kotlin 数据类型。
- 控制结构。
- 异常处理。

2.1　变　　量

在 Kotlin 语言中，定义了两种变量类型：var 或 val。其中，var 表示为可变引用（具备读—写功能），并可在初始化后更新；var 关键字可用于定义 Kotlin 中的变量，也等价于常规的 Java 变量。如果变量在某一时刻发生变化，应采用 var 关键字对其进行声明。下列示例展示了变量的声明过程：

```
fun main(args: Array<String>) {
    var fruit:String = "orange" // 1
    fruit = "banana" // 2
}
```

其中：

- 定义 fruit 变量，并利用 orange 变量值对其进行初始化。
- 利用 banana 值再次初始化 fruit 变量。

第二种变量类型表示为只读引用，该变量类型在初始化后无法重新赋值。



val 关键字可包含自定义 getter 方法，因而从技术角度上讲，可在每次访问后返回不同的对象。换而言之，无法保证底层对象引用是不可变的。

```
val random: Int
get() = Random().nextInt()
```

第 4 章将深入讨论自定义 getter 方法。

val 关键字等同于包含 final 修饰符的 Java 变量。不可变变量的功效在于，可确保变量不被错误地修改。另外，当与多线程协同工作时，不可变性同样十分有用，且不必担心相应的数据同步操作。当声明不可变变量时，可使用 val 关键字，如下所示：

```
fun main(args: Array<String>) {
    val fruit:String= "orange"// 1
    fruit = "banana" // 2 Error
}
```

- 定义 fruit 变量，并利用 orange 字符串值对其进行初始化。
- 此时编译器将抛出一条错误——fruit 变量已经被初始化。



Kotlin 还可定义文件级别的变量和函数，详细内容将在第 3 章讨论。

需要注意的是，变量引用类型（var，val）与其自身引用相关，而不是引用对象的属性。这也表明，当使用只读引用（val）时，将无法改变指向特定对象实例的引用（无法再向变量赋值），但仍可修改引用对象的属性。下面通过数组方式对此加以考察。

```
val list = mutableListOf("a","b","c") // 1
list = mutableListOf("d", "e") // 2 Error
list.remove("a") // 3
```

- 初始化可变列表。
- 此时编译器将抛出一条错误——值引用无法被修改（重新赋值）。
- 编译器允许修改当前列表中的内容。

关键字 val 无法保证底层对象不可变。

如果须保证当前对象不可被修改，则应使用不可变引用以及一个不可变对象。然而，Kotlin 标准库包含了一个集合接口的不可变等价结构（List 和 MutableList，Map 和 MutableMap 等）；同样，对于创建特定集合实例的帮助函数也是如此，如表 2.1 所示。

表 2.1

变量/值定义	引用是否可变	对象状态是否可变
val = listOf(1,2,3)	否	否
val = mutableListOf(1,2,3)	否	是
var = listOf(1,2,3)	是	否
var = mutableListOf(1,2,3)	是	是

2.2　类型推断

通过上述示例可以看出，Kotlin 类型定义在变量名称之后，这一点与 Java 不同，如下所示：

```
var title: String
```

初看之下，Java 程序员可能会感到奇怪，但这种定义方式则是 Kotlin 中非常重要的特征之一，即类型推断。类型推断意味着，编译器可从上下文中推断类型（赋予某个变量的表达式值）。当变量声明和初始化工作一起执行时，即可省略类型声明。考察下列变量定义：

```
var title: String = "Kotlin"
```

其中，变量 title 的类型为 String，但是否真正需要一个隐式类型声明以确定变量类型？表达式右侧表示为字符串 Kotlin，可将其赋予表达式左侧所定义的变量 title。

显然，此处将变量类型定义为 String，这与赋值表达式（Kotlin）具有相同类型。同样，Kotlin 编译器也知晓这一事实，因此当声明某个变量时，可省略相关类型。针对源自当前上下文的变量，编译器会尝试确定最佳类型，如下所示：

```
var title = "Kotlin"
```

需要注意的是，此处虽省略了类型声明，但变量类型仍隐式地设置为 String——Kotlin 是一种强类型语言。这也解释了上述两项声明彼此相同的原因，Kotlin 编译器仍可验证全部变量的特征应用。相关示例如下所示：

```
var title = "Kotlin"
title = 12 // 1, Error
```

其中，推断类型为 String，而此处尝试赋予 Int。

如果希望像 title 变量赋值 Int（对应值为 12），则须将 title 类型定义为 String 和常见类

型 Int，在类型体系结构中，最为接近的结果是 Any，如下所示：

```
var title: Any = "Kotlin"
title = 12
```

Any 等价于 Java 中的对象类型，同时也是 Kotlin 类型体系结构中的根节点，如图 2.1 所示。Kotlin 中的全部类显式地继承自 Any 类型，甚至是 String 或 Int 等基本类型。

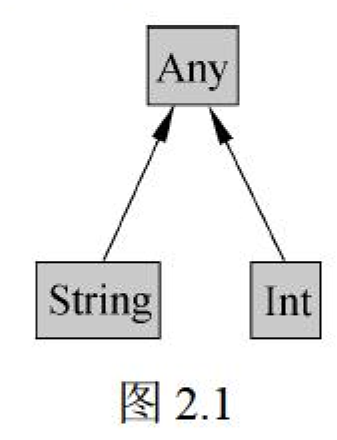


图 2.1

Any 定义了 3 种方法，分别是 equals、toString 和 hashCode。Kotlin 标准库针对该类型设置了一些扩展，第 7 章将对此加以讨论。

不难发现，类型推断不仅限于基本类型，下面考察直接源自函数的推断类型。

```
var total = sum(10, 20)
```

在该示例中，推断类型等同于函数的返回类型。此处可猜测为 Int 类型，但也可能是 Double、Float 或其他类型。如果难以从上下文中推断相应类型，则可单击变量名，并运行 Android Studio 表达式类型命令（在 Windows 中，为 Shift+Ctrl+P 快捷键；在 macOS 中，为箭头+control+P），并以工具提示方式显示相应的变量类型，如图 2.2 所示。

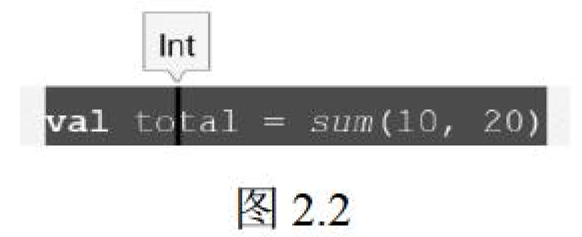


图 2.2

类型推荐机制同样适用于泛型，如下所示：

```
var persons = listOf(personInstance1, personInstance2)
// Inferred type: List<Person> ()
```

假设仅传递 Person 类实例，则推断类型为 List<Person>。另外，listOf 方法是定义于 Kotlin 标准库中的帮助函数，并可创建集合，第 7 章将对此加以讨论。下面考察相对高级的示例，并使用称之为 Pair 的 Kotlin 标准库类型，其中包含了由两个数值组成的二元对，如下所示：

```
var pair = "Everest" to 8848 // Inferred type: Pair<String, Int>
```

其中，pair 实例通过中缀函数创建，第 4 章将对此进行解释。当前，读者仅须了解两个声明均返回相同的 Pair 对象类型，如下所示：

```
var pair = "Everest" to 8848
// Create pair using to infix method
var pair2 = Pair("Everest", 8848)
// Create Pair using constructor
```

类型推断还适用于更为复杂的场合，例如从推断类型中进行类型的推断。下面使用 Kotlin 标准库中的 mapOf 函数，以及 Pair 类中的 to 方法定义 map。其中，二元对中的第一项用于推断 map 键类型；第二项用于推断值类型，如下所示：

```
var map = mapOf("Mount Everest" to 8848, "K2" to 4017)
// Inferred type: Map<String, Int>
```

Map<String, Int>的泛型根据 Pair<String, Int>进行推断，也就是说，根据传递至 Pair 构造方法中的参数类型进行推断。这里，读者可能会产生疑问：如果用于创建 map 的二元对彼此不同，情况又当如何？此处，第一个二元对表示为 Pair<String,Int>，第二个二元对表示为 Pair<String, String>，如下所示：

```
var map = mapOf("Mount Everest" to 8848, "K2" to "4017")
// Inferred type: Map<String, Any>
```

其中，Kotlin 编译器针对全部二元对尝试推断基本类型。两个二元对中的第一个参数表示为 String（Mount Everest，K2）。自然地，此处推断类型为 String。二元对中的第二个参数则有所不同（第一个二元对为 Int，第二个二元对为 String）。因此，Kotlin 须获取最近的基本类型。由于上行类型体系结构中最近的基本类型为 Any，因而此处选择了 Any 类型，如图 2.3 所示。

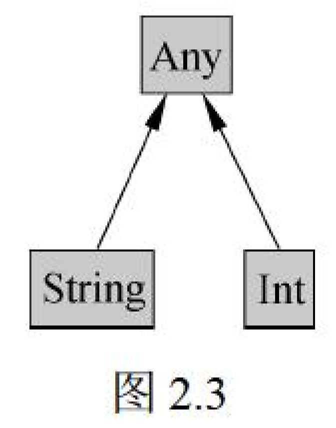


图 2.3

不难发现，类型推断机制在大多数时候工作良好，但必要时，仍可选择显式定义某种数据类型。例如需要定义不同的变量类型时：

```
var age: Int = 18
```

当处理整数时，Int 类型通常是默认选择，但依然可显式地定义不同类型，例如 Short，进而节省 Android 内存空间，如下所示：

```
var age: Short = 18
```

另外一方面，如果希望存储较大的数值，可将 age 的变量类型定义为 Long。如前所述，可采用显式类型声明，或者使用字面常量，如下所示：

```
var age: Long = 18 // Explicitly define variable type
var age = 18L
// Use literal constant to specify value type
```

上述两项声明具有相同效果，均可生成 Long 类型的变量。

代码中常会遇到省略类型声明，以使语法更为简洁。但在某些场合下，由于上下文环境缺失，Kotlin 编译器无法进行类型推断。例如，不包含赋值行为的简单声明往往会导致类型推断失败，如下所示：

```
val title // Error
```

其中，变量将于后续操作中进行初始化，因而目前尚无法确定其类型，这也是类型须显式定义的原因。一般的规则是，如果表达式类型针对编译器已知，则可对类型进行推断；否则，需要对其进行显式定义。Android Studio 中的 Kotlin 在这一方面表现得较好，并可对类型推断错误之处予以高亮显示。也就是说，在编写代码时，IDE 可即时显示相应的错误信息。

2.3　严格的空保护机制

根据敏捷软件评估（对应网址为 http://p3.snf.ch/Project-144126）结果显示，在 Java 系统中，空安全检测是最为常见的问题之一。在 Java 中，最大的问题在于 NullPointer Exception。

为了避免 NullPointerException，需要编写相应的防护代码，并检测某一对象在使用前是否为空。大多数现代编程语言通过相关步骤将运行期错误转化为编译期错误，例如 Kotlin 语言。在 Kotlin 中，一种方式是向语言类型系统中添加空保护机制。Kotlin 类型系统可辨识空引用以及非空引用。该特性在开发的各个阶段均可检测与 NullPointer Exception 相关的错误。编译器以及 IDE 均会阻止 NullPointerException。在大多数时候，程序将产生编译期错误，而非运行期错误。

空保护机制隶属于 Kotlin 的类型系统。默认状态下，常规类型通常无法设置为空值（无法存储空引用），除非对其予以显式定义。当存储空引用时，需要标记可空变量（可存储

空引用)，即在变量类型声明中添加问号，如下所示：

```
val age: Int = null // 1, Error
val name: String? = null // 2
```

针对第一行代码，编译器将抛出错误消息——该类型不可为空。在第二行代码中，可执行空赋值操作——该类型采用问号后缀标记为可空。

对于潜在空对象，将无法调用其中的方法，除非在调用前进行检测，如下所示：

```
val name: String? = null
//...
name.toUpperCase() // error, this reference may be null
```

稍后将讨论如何解决此类问题。Kotlin 语言中的非空类型均包含空类型的等价形式，例如 Int 和 Int?，String 和 String?等。同样的规则也适用于 Android 框架中的所有类（例如 View 和 View？）。这也表明，各种非泛型类均可用于定义两种类型，即空类型和非空类型。同时，非空类型也可表示为其空等价形式的子类型。例如，Vehicle（也是 Vehicle?的子类型）也可表示为 Any 的子类型，如图 2.4 所示。

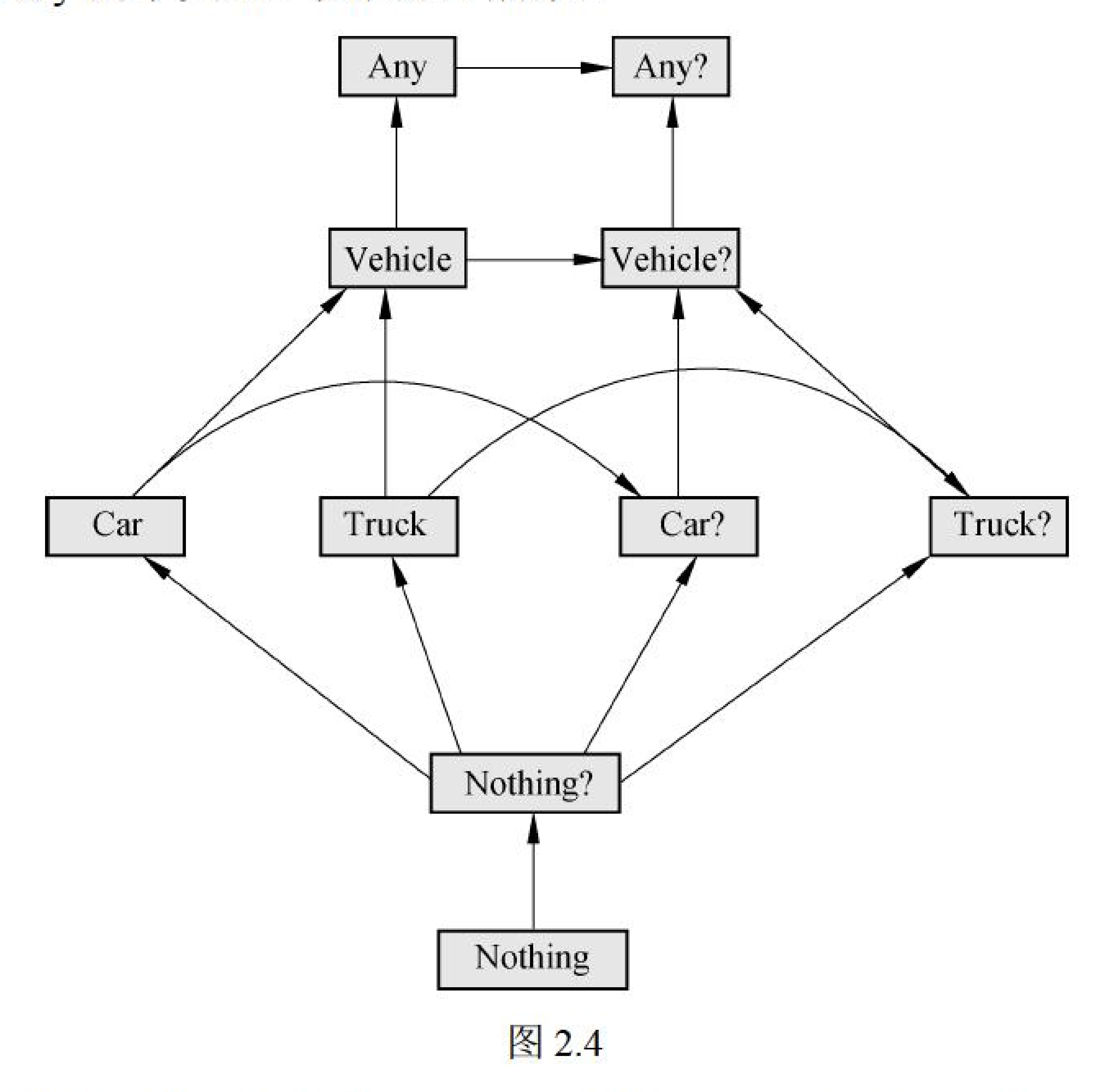


图 2.4

其中，Nothing 类型表示为一种空类型，且无法生成相应的实例。第 3 章将对此加以深入讨论。根据这一类型体系结构，可将非空对象（Vehicle）赋予至可空类型变量（Vehicle?），但却不可将可空对象（Vehicle?）赋予至非空变量（Vehicle），如下所示：

```
var nullableVehicle: Vehicle?
var vehicle: Vehicle

nullableVehicle = vehicle // 1
vehicle = nullableVehicle // 2, Error
```

在第 3 行代码中，可执行赋值操作；而在第 4 行代码中，由于 nullableVehicle 可能为空，因而将会产生错误。

后续章节将继续探讨可空类型，下面再次返回类型定义。当定义泛型时，存在多种可空类型的定义方式。对于泛型 ArrayList（其中包含了 Int 类型的数据项），表 2.2 通过比较不同的声明方式，检测各种集合类型。

表 2.2

类型声明	表是否可为空	元素是否可为空
ArrayList<Int>	否	否
ArrayList<Int>?	是	否
ArrayList<Int?>	否	是
ArrayList<Int?>?	是	是

由于编译器会强制消除 NullPointerException，因此读者应深入理解空类型声明的不同方式。这也表明，在访问潜在的空引用之前，编译器将执行空检测。下面考察 Activity 类 onCreate 中的常见 Android/Java 错误。

```
// Java
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    savedInstanceState.getBoolean("locked");
}
```

在 Java 中，代码将成功编译，访问空对象将导致应用程序在运行期内崩溃，同时抛出 NullPointerException。下面查看该方法的 Kotlin 版本。

```
override fun onCreate(savedInstanceState: Bundle?) { // 1
     super.onCreate(savedInstanceState)
     savedInstanceState.getBoolean("key") // 2 Error
}
```

其中，savedInstanceState 定义为 Bundle?。此时，编译器将抛出错误消息。

savedInstanceState 是一种平台类型，Kotlin 可将其解释为空类型或非空类型。稍后将

讨论平台类型，当前仅将 savedInstanceState 定义为可空变量——首次创建 Activity 时，将会传递空值。当采用保存后的实例状态再次创建 Activity 时，仅传递 Bundle 实例。



第 3 章将讨论函数。目前，Kotlin 中函数的声明方式与 Java 十分类似。

在 Kotlin 中，空类型检测方式与 Java 基本相同。

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
}

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    val locked: Boolean
    if(savedInstanceState != null)
        locked = savedInstanceState.getBoolean("locked")
    else
        locked = false
}
```

在 Java 开发过程中，空检测是一类十分常见的操作（特别是在 Android 框架中，其中大部分元素均为可空）。然而，Kotlin 支持更为简单的操作，进而处理可空变量，安全调用便是其中之一。

2.3.1　安全调用

安全操作符表示为一个问号和“.”的组合。注意，安全转换操作符总是返回某个值。如果操作符左侧为 null，则返回 null；否则将返回右侧表达式结果。

```
override fun onCreate(savedInstanceState: Bundle?) {
     super.onCreate(savedInstanceState)
     val locked: Boolean? = savedInstanceState?.getBoolean("locked")
}
```

如果 savedInstanceState 为 null，则结果返回 null；否则，将返回 savedInstanceState?.getBoolean("locked")表达式的结果。需要注意的是，可空引用调用总是返回可空结果，因此，表达式整体结果表示为可空的 Boolean?。若希望获得非空的布尔值，则可将安全调用操作符和 elvis 操作符结合使用，稍后将对此加以讨论。

多次调用安全调用操作符可形成链式效果，从而可避免使用内嵌 if 表达式或一些较为复杂的条件，如下所示：

```
// Java idiomatic - multiple checks
val quiz: Quiz = Quiz()
//...
val correct: Boolean?

if(quiz.currentQuestion != null) {
    if(quiz.currentQuestion.answer != null ) {
        // do something
    }
}
// Kotlin idiomatic - multiple calls of save call operator
val quiz: Quiz = Quiz()

//...

val correct = quiz.currentQuestion?.answer?.correct
// Inferred type Boolean?
```

其中，链式效果的工作方式可描述为：仅当 answer 值非 null，方可访问 correct；仅当 currentQuestion 值非 null，answer 方可被访问。最终，表达式将返回 correct 属性的返回值；或者，如果安全调用链中的任意对象均为 null，则返回 null。

2.3.2　elvis 操作符

elvis 操作符采用问号和冒号组合显示，即“?:”，对应语法显示如下所示：

```
first operand ?: second operand
```

其工作方式可描述为：如果第一个操作数不为 null，则返回该操作数；否则返回第二个操作数。通过 elvis 操作符，读者可编写更为简洁的代码。

当使用 elvis 操作符时，将获得非空的 locked 变量，如下所示：

```
override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
val locked: Boolean = savedInstanceState?.getBoolean("locked") ?: false
}
```

在上述代码中，如果 savedInstanceState 非空，elvis 操作符将返回 savedInstanceState?.getBoolean("locked")表达式；否则，将返回 false。该方式可确保正确处理 locked 变量。根

据 elvis 操作符，还可定义默认值。另外，仅当左侧结果为 null，方进一步计算右侧表达式。随后，当表达式为空时，将提供相应的默认值。对此，可适当调整 2.3.1 节中的示例代码，以使其总是返回非空值，如下所示：

```
val correct = quiz.currentQuestion?.answer?.correct ?: false
```

最终，表达式将返回 correct 属性的返回值；或者，如果安全调用链中的任何对象为 null，则结果返回 false。这也意味着，对应值总会被返回，因此推断结果表示为非空的布尔值。



elvis 操作符的名称源自美国著名歌手 Elvis Presley（埃尔维斯·普莱斯利），其发型像是一个大大的问号，如图 2.5 所示。

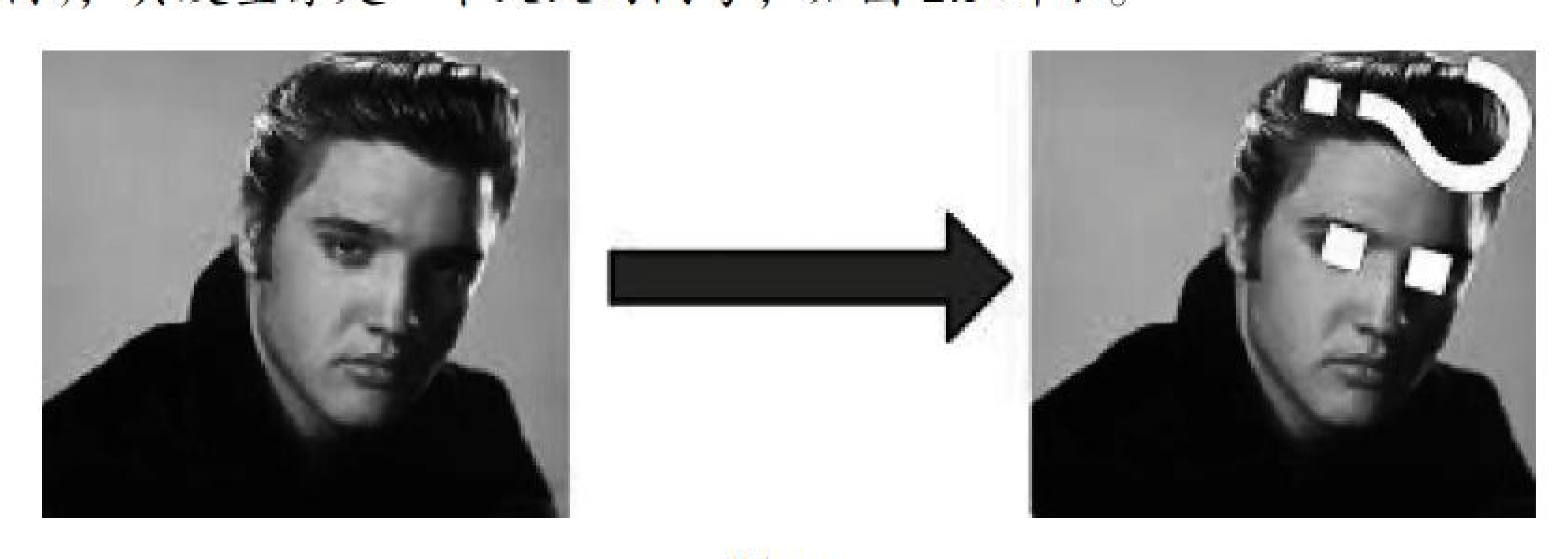


图 2.5

2.3.3　非空断言

非空断言则是另一种空检测工具，并采用两个问号表示。该操作符显式地将可空变量转换为非空变量，如下所示：

```
var y: String? = "foo"
var size: Int = y!!.length
```

正常情况下，无法将某个值从可空属性 length 赋予非空变量 size 中。然而，此处可明确告知编译器：当前可空变量可包含一个值。如果这一结论正确，应用程序即可以正确方式工作；否则，变量将包含 null 值，应用程序将抛出 NullPointerException。下面考察 onCreate 方法中的对应操作。

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    val locked: Boolean = savedInstanceState!!.getBoolean("locked")
}
```

上述代码可进行编译，但是否可正常地工作？如前所述，当恢复某个操作实例时，savedInstanceState 将传递至 onCreate 方法中。因此，当前代码正常工作且不会抛出异常。

然而，若生成操作实例时 savedInstanceState 为 null（不存在之前的实例可供恢复），因而将在运行期内抛出 NullPointerException。这一行为与 Java 类似，二者间的主要差别在于，在 Java 中，访问潜在可空对象且未经过空检测可视为一种默认行为；而在 Kotlin 中，则须对此予以强制执行；否则将产生编译错误。

该操作符仅存在少量的正确应用场合。因此，当在代码中使用或查看非空断言时，须注意其潜在的危险和警告。此处建议，应尽量减少其使用频率，在大多数时候，可采用安全调用智能转换。



http://bit.ly/2xg5JXt 中提供了少量的可用示例。其中，非空断言操作符被其他操作符所替代，进而保证正确的 Kotlin 结构。

实际上，上述示例并无必要使用非空断言操作符；相反，可通过更加安全的方式处理这一问题，即 let。

2.3.4　let

let 是另一种可空变量的处理方式。实际上，let 并非是一类操作符，同时也不是一种特定的语言结构。let 是 Kotlin 标准库中所定义的一个函数。下面结合安全调用操作符对 let 的语法加以考察。

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    savedInstanceState?.let{
        println(it.getBoolean("isLocked")) // 1
    }
}
```

let 中的 savedInstanceState 可通过命名变量进行访问。

如前所述，如果左侧 Eric 非 null，则可计算安全调用操作符的右侧表达式。在当前示例中，右侧表示为 let 函数，并接收另一个函数（Lambda）作为参数。当 savedInstanceState 不为 null 时，将执行 let 之后的代码块中的代码。关于函数，第 7 章将对此加以介绍。

2.4　可空性和 Java

前述内容讨论了 Kotlin 中须显式定义包含 null 值的引用。另外一方面，Java 对此则并不十分严格。这里的问题是，Kotlin 如何处理来自 Java 中的数据类型（基本上讲，全部 Android SDK 和库均采用 Java 编写）？无论如何，Kotlin 编译器都将根据代码确定类型的

可空性，并通过可空标注，将类型表示为可空或非空类型。

Kotlin 编译器支持多种可空标注形式，其中包括：

- Android（com.android.annotations 和 android.support.annotations）。
- JetBrains（源自 org.jetbrains.annotations 包的@Nullable 和@NotNull）。
- JSR-305（Javax.annotation）。

读者可访问 https://github. com/ JetBrains/ kotlin/ blob/ master/ core/ descriptor. loader.Java/src/ org/jetbrains/kotlin/load/Java/JvmAnnotationNames.kt 以查看 Kotlin 编译器的完整源代码。

在 Activity 类的 onCreate 方法中，savedInstanceState 类型显式地设置为可空类型 Bundle?，如下所示：

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
}
```

然而，也存在某些场合无法确定变量的可空性。除了标记为不可空类型之外，源自 Java 的全部变量均可为 null。相应地，可将其全部视为可空类型，并在每次访问之前予以检测，但该行为缺乏实际操作意义。针对这一问题，Kotlin 引入了平台类型这一概念，且源于 Java 类型，并包含了相对宽松的 null 检查。这也意味着，平台类型可以为 null 或非空。

尽管无法亲自声明平台类型，这一特殊语法的存在原因在于，编译器和 Android Studio 某些时候需要对其加以显示。对此，可将平台类型置于异常消息或方法参数列表中。在变量类型声明中，平台类型语法仅表示为一个单一的感叹号后缀，如下所示：

```
View! // View defined as platform type
```

各种平台类型均可视为可空类型，但类型的可空机制通常取决于上下文。因此，某些时候可将其视为非空变量。下列伪代码显示了平台类型的可能含义。

```
T! = T or T?
```

开发人员负责处理如何定义相关类型（可空或不可空）。下面考察 findViewById 方法中的具体应用。

```
val textView = findViewById(R.id.textView)
```

对此，相关问题包括：findViewById 方法的实际返回内容；textView 的推断类型；可空类型（TestView）或是不可空类型（TestView?）？默认状态下，Kotlin 编译器对于 findViewById 方法返回值的可空性一无所知。这也是 TextView 的推断类型包含平台类型 TextView!的原因。

由于程序员了解布局中是否包含所配置的 TextView（横向、纵向等），因而应负责处理此类事项，或其中的部分内容。若在当前布局中定义了适宜的视图，findViewById 方法将返回该视图的引用，否则将返回 null，如下所示：

```
val textView = findViewById(R.id.textView) as TextView // 1
val textView = findViewById(R.id.textView) as TextView? // 2
```

针对第一行代码，假设 textView 出现于每项配置的各个布局中，因此 textView 可定义为不可空类型。在第二行代码中，假设 textView 未出现于全部布局配置中（例如仅出现于横向模式中），textView 须定义为可空类型；否则，当尝试向某一非空变量赋值 null 时（载入未包含 textView 的布局），将会抛出 NullPointerException。

2.5 转　换

大多数编程语言均支持数据转换这一概念。基本上讲，转换是指将某一特定类型的对象转换为另一种类型。在 Java 中，需要在进行访问前显式地转换对象；或者转换后将其存储在转换类型的变量中。Kotlin 则简化了转换概念，并引入了智能转换这一概念。

在 Kotlin 中，可执行某些类型转换操作，其中包括：

- 显式地将对象转换为不同的类型（安全转换操作符）。
- 将对象转换为不同类型；或者将可空类型隐式地转换为非空类型（智能转换机制）。

2.5.1 安全/不安全转换操作符

在强类型语言中，例如 Java 或 Kotlin，需要显式地采用转换操作符，并将数值从一种类型转换为另一种类型。典型的转换操作是通过某种特定的对象，将其转换为另一种对象类型，包括超类型（向上转换）、子类型（向下转换）以及接口。下面考察 Java 中的转换示例：

```
Fragment fragment = new ProductFragment();
ProductFragment productFragment = (ProductFragment) fragment;
```

在上述代码中，ProductFragment 实例被赋予存储 Fragment 数据类型的变量。为了将该变量赋予至仅可存储 ProductFragment 数据类型的 productFragment 变量中，则需要执行显式转换。与 Java 不同，Kotlin 设置了特定的关键字 as 作为非安全转换操作符，进而处理转换问题，如下所示：

```
val fragment: Fragment = ProductFragment()
val productFragment: ProductFragment = fragment as ProductFragment
```

ProductFragment 表示为 Fragment 的子类型，因而上述示例工作正常。这里的问题是，不兼容类型转换将会抛出 ClassCastException 异常，这也是 as 操作符称作非安全转换操作符的原因，如下所示：

```
val fragment : String = "ProductFragment"
val productFragment : ProductFragment = fragment as
    ProductFragment
// Exception: ClassCastException
```

针对这一问题，可使用安全转换操作符或 as?，有时也称作可空转换操作符。该操作符尝试将某一值转换为特定类型，如果该值无法被转换，则返回 null，如下所示：

```
val fragment: String = "ProductFragment"
val productFragment: ProductFragment? = fragment as?
    ProductFragment
```

注意，使用安全转换操作符需要将 name 变量定义为可空类型（即 ProductFragment?，而非 ProductFragment）。作为替代方案，可使用非安全转换操作符以及可空类型 Product Fragment?，因而可实际查看到所转换的类型，如下所示：

```
val fragment: String = "ProductFragment"
val productFragment: ProductFragment? = fragment as
    ProductFragment?
```

如果定义非空类型的 productFragment 变量，则需要使用 elvsi 操作符赋予默认值，如下所示：

```
val fragment: String = "ProductFragment"
val productFragment: ProductFragment? = fragment as?
    ProductFragment ?: ProductFragment()
```

当前，fragment as? ProductFragment 表达式将正确地被计算。如果该表达式返回非空值（也就是说，可执行转换），则对应值将赋予 productFragment 变量；否则，默认值（Product Fragment 的新实例）将被赋予至 productFragment 变量中。两种操作符之间的比较结果如下所示。

- 非安全型转换（as）：当转换无法实现时，抛出 ClassCastException 异常。
- 安全型转换（as?）：当转换无法实现时，返回 null。

当理解了非安全型转换与安全型转换之间的差异后，即可安全地从 fragment 管理器中获得 fragment，如下所示：

```
var productFragment: ProductFragment? = supportFragmentManager
.findFragmentById(R.id.fragment_product) as? ProductFragment
```

安全型转换和非安全型转换操作符可用于转换复杂对象。当与基本类型协同工作时，可简单地使用某种 Kotlin 标准库转换方法。Kotlin 标准库的大多数对象中均包含了相关方法，可简化基本的类型转换操作。转换过程可描述为：此类函数设置了前缀 to，以及需要转换的类名。在下列代码示例中，Int 类型通过 toString 方法转换为 String 类型。

```
val name: String
    val age: Int = 12
    name = age.toString(); // Converts Int to String
```

下面讨论基本类型及其转换操作。

2.5.2　智能转换

智能转换将某种类型的变量转换为另一种类型。与安全型转换不同，该转换行为通过隐式方式执行（无须使用 as 或 as?操作符）。只有当 Kotlin 编译器完全确定变量在检查后不会更改时，智能转换才会起作用。对于多线程应用程序，这体现了一种更为安全的操作方式。通常情况下，智能转换适用于全部不可变引用以及本地可变引用。智能转换包含两种类型，如下所示。

- 类型智能转换：将某种类型的对象转换为另一种类型的对象。
- 可空型类型转换：将可空型引用转换为非空型引用。

（1）类型智能转换

下面考察之前提到的 Animal 和 Fish 类，如图 2.6 所示。

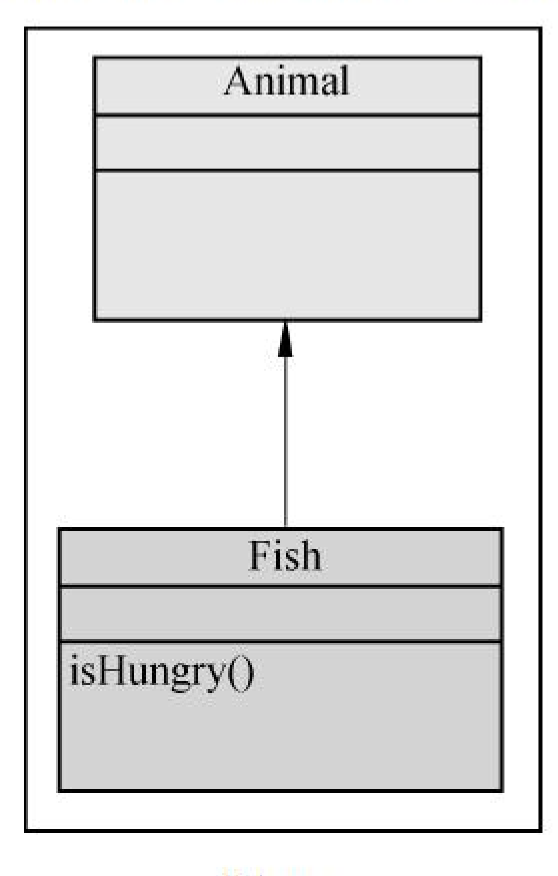


图 2.6

假设须调用 isHungry 方法，并检测 animal 是否为 Fish 的实例。在 Java 中，可执行下列操作：

```
// Java
if (animal instanceof Fish){
    Fish fish = (Fish) animal;
    fish.isHungry();
    // or
    ((Fish) animal).isHungry();
}
```

上述代码的问题主要是冗余性。代码需要检测 animal 实例是否为 Fish，随后显式地将 animal 转换为 Fish。那么，如果编译器能帮我们处理，岂不是更好?事实表明，当执行转换操作时，Kotlin 编译器具有足够的智能，并通过智能转换机制处理那些冗余转换。下列代码则是一个智能转换示例：

```
if(animal is Fish) {
    animal.isHungry()
}
```

Android Studio 中的智能转换



如果智能转换失败，Android Studio 将显示相应的错误消息，进而表明其可用性。当访问一个需要转换的成员时，Android Studio 将对应变量标记为绿色背景，如图 2.7 所示。

```
if(animal is Fish){
    animal.isHungry()
}
```

图 2.7

在 Kotlin 中，无须显式地将 animal 转换为 Fish，其原因在于，在类型检测后，Kotlin 编译器可隐式地处理转换。当前，在 if 代码块中，变量 animal 转换为 Fish。随后，最终结果与前述 Java 示例相同。因此，可安全地调用 isHungry 方法，且无须进行显式转换。需要注意的是，在当前示例中，智能转换的范围仅限于 if 代码块，如下所示：

```
if(animal is Fish) {
    animal.isHungry() // 1
}

animal.isHungry() // 2, Error
```

在第二行代码中，animal 示例表示为 Fish，因而可调用 isHungry 方法。在第 4 行代码中，animal 实例仍为 Animal，因而无法调用 isHungry 方法。

有些时候，智能转换的操作范围将会超出某个代码块，如下所示：

```
val fish:Fish? = // ...
if (animal !is Fish) // 1
    return

animal.isHungry() // 1
```

在最后一行代码中，animal 隐式地转换为非空型 Fish。

在上述示例中，如果 animal 不是 Fish，整体方法将从函数中返回。因此，编译器在代码块的其余范围内知晓 animal 定为 Fish。Kotlin 和 Java 条件表达式均可实现延迟计算。

这也意味着，在表达式 condition1() && condition2()中，仅当 condition1 返回 true 时，方法 condition2 才被调用。这也是条件表达式右侧中使用智能转换类型的原因。

```
if (animal is Fish && animal.isHungry()) {
    println("Fish is hungry")
}
```

注意，若 animal 并非是 Fish，条件表达式的第二部分内容将不会被计算。当计算完毕后，Kotlin 即知晓 animal 即为 Fish（智能转换）。

（2）非空型智能转换

智能转换还可处理其他情形，包括空检测。此处，假设 view 变量标记为可空类型——当前尚不知晓 findViewById 是否返回 view 或 null。

```
val view: View? = findViewById(R.layout.activity_shop)
```

安全操作符可能用于访问 view 中的相关方法和属性。在某些场合下，可能需要在同一对象上执行更多操作。此时，智能转换将是一种较好的解决方案，如下所示：

```
val view: View?

if ( view != null ){
    view.isShown()
    // view is casted to non-nullable inside if code block
}

view.isShown() // error, outside if the block view is nullable
```

当按照上述方式执行空检测时，编译器自动将可空型 view（View?）转换为非空型（View）。因此，可在 id 代码块中调用 isShown 方法，且无须使用安全调用操作符。在 if 代码块之外，view 仍为可空类型。

智能转换仅与只读类型变量协同工作——读-写变量将在执行检测和访问变量时发生改变。

除此之外，智能转换还可用于函数的返回语句。如果在包含返回语句的函数中执行空检测，那么变量也将转换为非空类型，如下所示：

```
fun setView(view: View?){
    if (view == null)
    return
    // view is casted to non-nullable
    view.isShown()
}
```

其中，Kotlin 知晓当前变量值不为 null；否则将执行 return 语句。第 3 章将对函数问题加以讨论。通过 elvis 操作符，此处可采用前述更加简单的语法，并通过一行代码执行空检测，如下所示：

```
fun verifyView(view: View?){
    view ?: return

    // view is casted to non-nullable
    view.isShown()
    //...
}
```

若不打算从函数中返回，并针对当前问题执行某些显式操作并抛出异常，则可结合 elvis 操作符和错误消息机制加以使用，如下所示：

```
fun setView(view: View?){
    view ?: throw RuntimeException("View is empty")
    // view is casted to non-nullable
    view.isShown()
}
```

综上所述，智能转换是一种功能强大的工具，并可有效地减少空检测的数量。因此，Kotlin 对其尤为推崇。对应的通用规则可描述为：仅当 Kotlin 知晓变量在转换后（甚至是另一个线程）无法被更改，智能转换方为有效。

2.6　基本数据类型

在 Kotlin 中，一切事物均表示为对象（引用类型，而不是基本类型）。也就是说，Kotlin 中不存在 Java 中的基本类型数据，这可有效地减少代码的复杂度，从而可在任意变量上调用相关方法和属性。例如，下列代码可将 Int 变量转换为 Char：

```
var code: Int = 75
code.toChar()
```

通常，诸如 Int、Long 或 Char 均经过适当优化（存储为基本类型），但仍可像其他对象那样调用相关方法。

默认状态下，Java 平台将数字存储为 JVM 基本类型，但当需要使用到可空型数字引用（例如 Int?）后调用泛型时，Java 将采用“装箱”（boxed）表达形式。“装箱”机制表明，基本元素将被封装至对应的“装箱”型基本类型中。这意味着，对应实例行为类似于一个对象。在 Java 中，基本类型的“装箱”型表达方式包括 int-Integer、long-Long 等。由于 Kotlin 代码将编译为 JVM 字节码，因而对应规则基本相同，如下所示：

```
var weight: Int = 12 // 1
var weight: Int? = null // 2
```

在第一行代码中，数值存储为基本类型；在第二行代码中，数值存储为“装箱”型整数（复合类型）。

因此，当每次生成某个数字（Byte、Short、Int、Long、Double、Float）、Chat 或 Boolean 时，将存储为基本类型，除非将其声明为可空类型（Byte?、Char?、Array?等）；否则，对应数据将存储为“装箱”型表达方式，如下所示：

```
var a: Int = 1 // 1
var b: Int? = null // 2
b = 12 // 3
```

在第一行代码中，a 定义为非空类型，因而存储为基本类型；在第二行代码中，b 定义为 null，因而存储为“装箱”型表达方式；在第三行代码中，b 仍存储为“装箱”型表达方式，尽管该变量包含了一个具体值。

泛型数据无法通过基本类型予以参数化，因而可采用“装箱”机制。需要注意的是，采用这一类封装机制（而非基本类型）可能会对性能产生影响，与基本类型表达方式相比，前者将产生内存开销。对于包含大量数据元素的表和数组来讲，这种影响尤为明显。因此，当对应用程序性能要求较高时，应采用基本类型数据。另外一方面，对于单变量，甚至是多变量声明，则无须担心数据的表达类型，即使在 Android 这一类对内存空间要求较高的设备上也是如此。

下面将讨论 Kotlin 中较为重要的基本数据类型，包括数字、字符、布尔值以及数组。

2.6.1　数字

Kotlin 中的基本数字类型等同于 Java 中的数字类型，如图 2.8 所示。

Type	Bit width
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

图 2.8

尽管如此，Kotlin 针对数字的处理方式与 Java 相比仍稍有不同。首先，数字间不存在隐式转换——较小的类型不会隐式地转换为较大的数据类型，如下所示：

```
var weight : Int = 12
var truckWeight: Long = weight // Error1
```

上述代码说明，在缺少显式转换的情况下，将无法将 Int 类型数值赋予 Long 变量。如前所述，在 Kotlin 中，一切事物均为对象，因而可调用相关方法，并显式地将 Int 类型转换为 Long 类型，进而修复当前问题，如下所示：

```
var weight:I nt = 12
var truckWeight: Long = weight.toLong()
```

上述代码类似于模板代码，但在实际操作过程中，可有效地避免与数字转换相关的诸多错误，同时节省大量的调试时间。相应地，Kotlin 标准库针对数字支持下列转换方法：

- toByte(): Byte。
- toShort(): Short。
- toInt(): Int。
- toLong(): Long。
- toFloat(): Float。
- toDouble(): Double。
- toChar(): Char。

然而，仍可显式地确定数字字面值，并调整为推断变量类型，如下所示：

```
val a: Int = 1
val b = a + 1 // Inferred type is Int
val b = a + 1L // Inferred type is Long
```

在数字方面，Kotlin 和 Java 之间的第二个差别是：在某些时候，数字的字面值稍有不同。对于整数值，存在下列多种字面常量：

```
27 // Decimals by default
27L // Longs are tagged by a upper case L suffix
0x1B // Hexadecimals are tagged by 0x prefix
0b11011 // Binaries are tagged by 0b prefix
```

Kotlin 并不支持八进制字面值，但会支持浮点数的常规表示法，如下所示：

```
27.5 // Inferred type is Double
27.5F // Inferred type is Float. Float are tagged by f or F
```

2.6.2　字符

Kotlin 的字符存储为 Char 类型。在大多数时候，字符与字符串相类似，因此本节主要考察二者间的相似性和差异。当定义 Char 时，须使用单引号，这一点与字符串的双引号有所不同，如下所示：

```
val char = 'a' // 1
val string = "a" // 2
```

在第一行代码中，定义了类型为 Char 的变量；第二行代码则定义了 String 类型的变量。

在字符和字符串中，可通过反斜杠表示转义字符，如下所示：

- \t：制表符。
- \b：退格键。
- \n：换行符。
- \r：换行符。
- \'：单引号。
- \"：双引号。
- \\：斜线符号。
- \$：美元字符。
- \u：Unicode 转义序列。

下列代码定义了包含 yinYang Unicode（U+262F）的 Char 类型：

```
var yinYang = '\u262F'
```

2.6.3　数组

在 Kotlin 中，数组表示为 Array 类。当在 Kotlin 中创建一个数组时，需要使用多个 Kotlin 标准库函数。其中，arrayOf()则是最为简单的函数，如下所示：

```
val array = arrayOf(1,2,3) // inferred type Array<Int>
```

默认状态下，该函数将创建 Int 数组。相应地，若打算定义 Short 或 Long 数组，则可显式地定义数组类型，如下所示：

```
val array2: Array<Short> = arrayOf(1,2,3)
val array3: Array<Long> = arrayOf(1,2,3)
```

如前所述，采用“装箱”式表达形式将降低应用程序的性能。因此，Kotlin 提供了某些特定类，以表示基本类型的数组，进而减少内存开销，即 ShortArray、IntArray、LongArray 等。尽管定义了系统的方法集和属性集合，但这些类与 Array 类并不包含继承关系。当创建类实例时，须使用相应的工厂函数，如下所示：

```
val array = shortArrayOf(1, 2, 3)
val array = intArrayOf(1, 2, 3)
val array = longArrayOf(1, 2, 3)
```

需要注意其中的某些差异——此类方法外观相似，但却生成不同类型的表达方式，如下所示：

```
val array = arrayOf(1,2,3) // 1
val array = longArrayOf(1, 2, 3) // 2
```

第一行代码表示 Long 数据元素的泛型数组（推断类型：Array<Long>）。第二行代码则表示包含基本 Long 数据元素的数组（推断类型：LongArray）。

知晓数组的实际尺寸往往会提升性能，因此，Kotlin 定义了另一个库函数 arrayOfNulls，并生成一个包含 null 元素的既定尺寸数组，如下所示：

```
val array = arrayOfNulls(3) // Prints: [null, null, null]
println(array) // Prints: [null, null, null]
```

除此之外，还可使用工厂函数填充预定义尺寸的数组，并使用数组尺寸作为第一个参数；并且 lambda 可以返回每个数组元素的初始值作为第二个参数，如下所示：

```
val array = Array (5) { it * 2 }
println(array) // Prints: [0, 2, 4, 8, 10]
```

第 5 章将详细讨论 lambda（匿名函数）。另外，Kotlin 中数组的访问方式与 Java 相同，如下所示：

```
val array = arrayOf(1,2,3)
println(array[1]) //Prints: 2
```

同时，数据元素的索引方式也与 Java 相同，也就是说，第一个元素的索引为 0，第二个元素的索引为 1，等等。在 Kotlin 中，数组是保持不变的，这一点与 Java 不同。相关内容将在第 6 章加以讨论。

2.6.4　布尔类型

布尔类型表示为一种逻辑类型，并包含两种可能值，即 true 和 false。除此之外，还可使用可空的布尔类型，如下所示：

```
val isGrowing: Boolean = true
val isGrowing: Boolean? = null
```

另外，布尔类型还支持大多数编程语言中的标准内建操作，如下所示：

- `||`：表示为逻辑 OR 操作。当两个判断式中的任意一个为 `true` 时则返回 `true`。
- `&&`：逻辑 AND 操作。当两个判断式均为 `true` 时则返回 `true`。
- `!`：否定运算符。针对 `false` 返回 `true`；而对 `true`，则返回 `false`。

注意，针对任意条件类型，仅可使用非空布尔值。

类似于 Java，仅在必要时||和&&方采用延迟计算。

2.7　复合数据类型

本节考察 Kotlin 中相对复杂的数据类型。与 Java 相比，某些数据类型得到了较为明显的改善，同时还包含了一些全新的数据类型。

2.7.1　字符串

Kotlin 字符串的行为方式与 Java 类似，但存在几点改进措施。当在特定索引处访问字符时，可使用索引操作符，并采用与访问数组元素相同的方式访问字符串，如下所示：

```
val str = "abcd"
println (str[1]) // Prints: b
```

另外，还可访问 Kotlin 标准库中的各种扩展，进而可方便地与字符串协同工作，如下所示：

```
val str = "abcd"
println(str.reversed()) // Prints: dcba
println(str.takeLast(2)) // Prints: cd
println("john@test.com".substringBefore("@")) // Prints: john
println("john@test.com".startsWith("@")) // Prints: false
```

这与 Java 中的 String 类保持一致，因而相关方法并不隶属于 String 类，而是定义为扩展。第 7 章将对此进行详细介绍。



读者可参考 String 类文档，以考察相关方法的完整列表（对应网址为 https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-string/）。

字符串的构建过程并不复杂，在 Java 中，通常需要使用到较长的连接表达式。在下列 Java 示例中，字符串由多个元素构建而成。

```
//Java
String name = "Eva";
int age = 27;
String message = "My name is" + name + "and I am" + age + "years old";
```

在 Kotlin 中，通过字符串模板，可极大地简化字符串的创建过程。对此，可不采用连接方式，并可通过美元符号（构造占位符）简单地将某个变量置于字符串中。在数据插入过程中，字符串占位符将被实际值替换，如下所示：

```
val name = "Eva"
val age = 27
val message = "My name is $name and I am $age years old"
println(message)
// Prints: My name is Eva and I am 27 years old
```

这可视作一种高效的连接方式，编译器代码生成一个 StringBuilder，并附加所有部分内容。同时，字符串模板并不仅限于单变量，还可涵盖“&{”和“}”字符之间的整体表达式。除此之外，还可表示为一个函数调用，并返回数值或属性访问，如下所示：

```
val name = "Eva"
val message = "My name has ${name.length} characters"
println(message) // Prints: My name has 3 characters
```

上述语法可创建更为简洁的代码，当变量或表达式中的某个值构造字符串时，无须打断该字符串。

2.7.2　范围

范围是数值序列的一种定义方式，并通过序列中的首、尾值表示。利用范围，可存储重量、温度、时间以及年龄等数据。范围可利用两个“.”予以表示（实际上是 rangeTo 操作符），如下所示：

```
val intRange = 1..4 // 1
val charRange= 'b'..'g' // 2
```

在第一行代码中，推断类型为 IntRange（对应形式为 i >= 1 && i <= 4）。在第二行代码中，推断类型为 CharRange（字母对应形式为从'b'至'g'）。



需要注意的是，应使用单引号定义字符范围。

Int、Long 和 Char 范围可用于在 for...each 循环中遍历下一个数值，如下所示：

```
for (i in 1..5) print(i) // Prints: 1234
for (i in 'b'..'g') print(i) // Prints: bcdefg
```

范围还可用于检测某一值大于初始值或小于终止值，如下所示：

```
val weight = 52
val healthy = 50..75

if (weight in healthy)
    println("$weight is in $healthy range")
    // Prints: 52 is in 50..75 range
```

同时还可针对其他范围类型采用这一方式，例如 CharRange。如下所示：

```
val c = 'k' // Inferred type is Char
val alphabet = 'a'..'z'

if(c in alphabet)
    println("$c is character") //Prints: k is a character
```

在 Kotlin 中，范围处于闭合状态（包括端值），这意味着，范围端值纳入至当前范围内，如下所示：

```
for (i in 1..1) print(i) // Prints: 123
```

注意，默认状态下，Kotlin 中的范围呈递增状态（默认步进值为 1），如下所示：

```
for (i in 5..1) print(i) // Prints nothing
```

当采用逆向遍历时，须使用 downTo 函数将步进值设置为-1，如下所示：

```
for (i in 5 downTo 1) print(i) // Prints: 54321
```

除此之外，还可设置不同的步进值，如下所示：

```
for (i in 3..6 step 2) print(i) // Prints: 35
```

注意，在 3..6 范围内，并不输出最后一个数值元素，其原因在于：步进索引在每次循环遍历时移动两个步长。因此，首次遍历后对应值为 3；第二次遍历后值为 5；最后，第三次遍历后值为 7——由于超出了当前范围，因而该值将被忽略。

step 定义的步长须为正值。如果设置了一个负值步长，则应将 downTo 函数与 step 函数协同使用，如下所示：

```
for (i in 9 downTo 1 step 3) print(i) // Prints: 963
```

2.7.3　集合

与集合协同工作可视为程序设计中的重要内容，相比于 Java，Kotlin 提供了多种类型的集合以及改进措施。第 7 章将对此进行深入讨论。

2.8　语句和表达式

与 Java 相比，Kotlin 提供了更为广泛的表达式应用空间，读者有必要了解语句和表达式之间的差别。程序基本上可表示为一个语句和表达式序列。其中，表达式生成某个值，并可作为另一个表达式、变量赋值或函数参数中的部分内容。另外，表达式可表示为一个或多个操作数（供操作的数据）以及一个或多个操作符形成的操作符（表示特定操作的符号），并可计算为某个单一值，如图 2.9 所示。

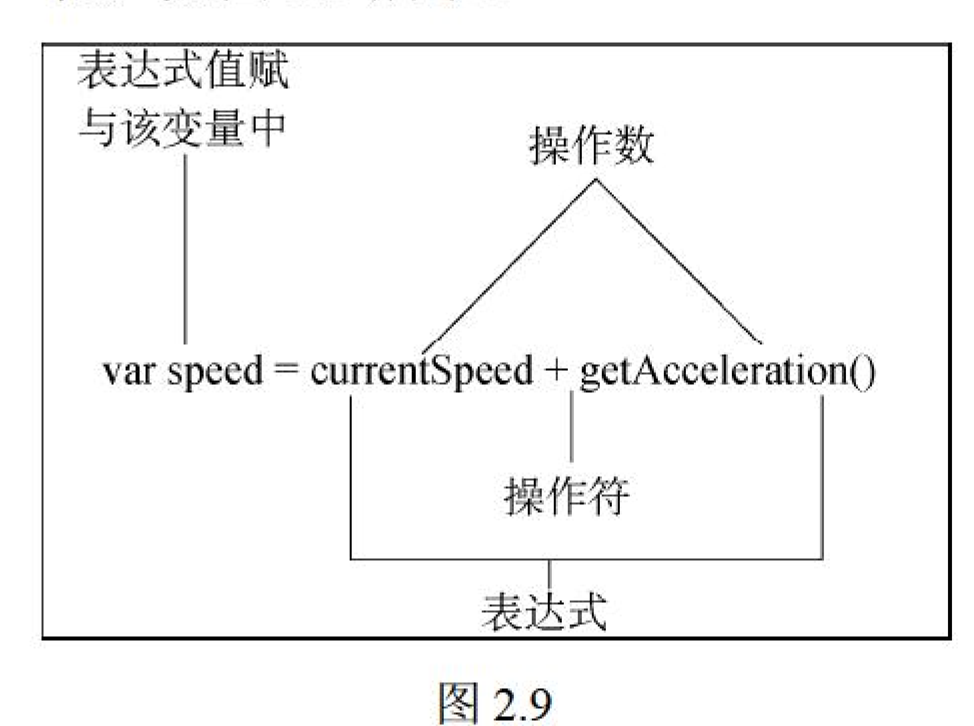


图 2.9

表 2.3 显示了 Kotlin 中的一些表达式示例。

表 2.3

表达式（生成某个值）	赋值	表达式类型
a = true	true	Boolean
a = "foo" + "bar"	"foobar"	String
a = min(2, 3)	2	Integer
a = computePosition().getX()	getX 的返回值	Integer

另外一方面，语句执行某项操作，且无法赋值于某个变量中。其中，语句可包含用于定义类（class）、接口（interface）、变量（val、var）、函数（fun）、循环逻辑（break 和 continue）等语言关键字。另外，当忽略表达式的返回值时，表达式也可视作语句（不可将值赋予变量，不可从函数中返回，不可将其用作其他表达式中的部分内容，等等）。

Kotlin 是一种面向表达式的语言。也就是说，Java 语言中的大多数结构可视作 Kotlin 中的表达式。实际上，Java 和 Kotlin 针对控制结构具有不同的看待方式。在 Java 中，控制结构表示为语句，而 Kotlin 则将其视为表达式（除了循环结构之外）。这也表明，Kotlin 可通过控制结构编写简洁的代码。稍后将考察具体示例。

2.9 控　制　流

Kotlin 包含了大量源自 Java 中的控制流，但提供了更加灵活、简便的操作方式。例如，Kotlin 中提供了 when 控制流结构，并以此替换 Java 中的 switch…case。

2.9.1 if 语句

Kotlin 中的 if 语句与 Java 具有同等作用，如下所示：

```
val x = 5

if(x > 10){
    println("greater")
} else {
    println("smaller")
}
```

在 Kotlin 中，if 代码块可包含一条语句或表达式，如下所示：

```
val x = 5

if(x > 10)
    println("greater")
else
    println("smaller")
```

在 Java 中，if 被视作为一条语句，而 Kotlin 则将其定义为一个表达式。这也是两种语言之间的主要差异之一；当然，Kotlin 使用了更为简洁的语法。例如，可将 if 表达式的结果直接传递至函数参数，如下所示：

```
println(if(x > 10) "greater" else "smaller")
```

由于 if 表达式计算后传递至 println 方法中，因而代码被压缩为一行。当条件 x > 10 为 true 时，表达式返回第一个分支（greater），否则表达式返回第二个分支（smaller）。下面考察另一个示例，如下所示：

```
val hour = 10
val greeting: String
if (hour < 18) {
    greeting = "Good day"
} else {
    greeting = "Good evening"
}
```

其中 if 用作一条语句。如前所述，Kotlin 中的 if 表示为一个表达式，该表达式的结果可赋予某个变量中。通过这一方式，可将 if 表达式结果直接赋予变量 greeting 中，如下所示：

```
val greeting = if (hour < 18) "Good day" else "Good evening"
```

但某些时候，if 语句分支中还须放置其他一些代码。对此，仍可将 if 用作表达式。随后，匹配 if 分支的最后一行代码将作为结果予以返回，如下所示：

```
val hour = 10
val greeting = if (hour < 18) {
    //some code
    "Good day"
} else {
    //some code
    "Good evening"
}

println(greeting) // Prints: "Good day"
```

如果使用 if 作为表达式而非语句，该表达式需要包含 else 分支。相比之下，Kotlin 版本更优于 Java。由于 greeting 变量定义为非空类型，编译器将验证 if 的整体表达式，检测过程涉及包含分支条件的全部情况。考虑到 if 可视作表达式，因而可将其用在字符串模板中，如下所示：

```
val age = 18
val message = "You are ${ if (age < 18) "young" else "of age" } person"
println(message) // Prints: You are of age person
```

与 Java 相比，将 if 视为表达式可获得更大的灵活性。

2.9.2　when 表达式

Kotlin 中的 when 表达式是一种多路分支语句。when 表达式旨在替代 Java 中的

switch...case 语句。与一系列的 if...else...if 相比，when 可视为一种更好的替代方案，同时可提供更加简洁的语法，例如：

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> println("x is neither 1 nor 2")
}
```

其中，when 表达式逐一与全部分支进行参数匹配，直至满足某个分支条件。该行为类似于 switch...case，但无须在每个分支后重复书写多个 break 语句。

类似于 if 子句，when 的使用方式包括忽略返回值的语句，以及赋值于变量中的表达式。若 when 用作表达式，满足分支最后一行的对应值将成为整体表达式值；若用作语句，对应值则简单地予以忽略。通常，如果不存在任何分支可满足当前条件，则计算 else 分支，如下所示：

```
val vehicle = "Bike"

val message= when (vehicle) {
    "Car" -> {
        // Some code
        "Four wheels"
    }
    "Bike" -> {
        // Some code
        "Two wheels"
    }
    else -> {
        //some code
        "Unknown number of wheels"
    }
}

println(message) //Prints: Two wheels
```

若分支中包含了多条指令，须将其置于由{...}定义的代码块中。当 when 用作表达式时（when 的计算值赋予变量中），各代码块的最后一行语句视作返回值。这一行为与 if 表达式相同。不难发现，这也是 Kotlin 结构中的常见行为，本书后续内容还将对此加以深入讨论。

若 when 用作表达式，则 else 须强制存在，除非编译器可证实分支条件覆盖全部情况。另外，当使用逗号时，还可在一个分支中处理多个匹配参数，如下所示：

```
val vehicle = "Car"

when (vehicle) {
    "Car", "Bike" -> print("Vehicle")
    else -> print("Unidentified funny object")
}
```

when 另一个较好的特性是可检测变量类型。Kotlin 可方便地验证特定类型的 is 或!is 值。此处，智能转换将再次派上用场——无须额外检测，即可在某个分支代码块中访问匹配类型的方法和属性，如下所示：

```
val name = when (person) {
    is String -> person.toUpperCase()
    is User -> person.name
    // Code is smart casted to String, so we can
    // call String class methods
    //...
}
```

类似地，when 还可检测包含特定值的对应范围和集合。下面使用 is 和!is 关键字，如下所示：

```
val riskAssessment = 47

val risk = when (riskAssessment) {
    in 1..20 -> "negligible risk"
    !in 21..40 -> "minor risk"
    !in 41..60 -> "major risk"
    else -> "undefined risk"
}

println(risk) // Prints: major risk
```

实际上，when 分支的右侧可放置任意类型的表达式，可以是方法调用或其他表达式。考察下列代码示例，其中，第二个 when 表达式用于 else 语句。

```
val riskAssessment = 80
val handleStrategy = "Warn"

val risk = when (riskAssessment) {
    in 1..20 -> print("negligible risk")
    !in 21..40 -> print("minor risk")
    !in 41..60 -> print("major risk")
    else -> when (handleStrategy){
```

```
        "Warn" -> "Risk assessment warning"
        "Ignore" -> "Risk ignored"
        else -> "Unknown risk!"
    }
}

println(risk) // Prints: Risk assessment warning
```

如前所述，when 是一种功能强大的结构，与 Java 的 switch 结构相比具有更多的控制行为，且不仅限于检测等式。除此之外，when 甚至还可替代 if...else if 语句链。如果 when 表达式中未包含任何参数，分支条件将类似于布尔表达式，并在对应条件为 true 时执行该分支，如下所示：

```
private fun getPasswordErrorId(password: String) = when {
    password.isEmpty() -> R.string.error field required
    passwordInvalid(password) -> R.string.error invalid password
    else -> null
}
```

上述全部示例均需要使用到 else 分支。当涉及全部可能情况时，可忽略某个 else 分支。下面考察基于布尔类型的简单示例：

```
val large:Boolean = true
when(large){
    true -> println("Big")
    false -> println("Big")
}
```

编译器可验证全部可能值将被处理，因而无须确定 else 分支。相同的逻辑也可应用于枚举结构和封装类中，第 4 章将对此加以讨论。

Kotlin 编译器负责执行检测，因而可确保各种情况均不会被遗漏。这也降低了 Java 中各种 bug 出现的概率。其中，开发人员往往会忘记处理 switch 语句中的某些问题（尽管多态通常是一类较好的解决方案）。

2.9.3　循环

作为一种控制结构，循环重复执行统一指令集，直至满足结束条件。在 Kotlin 中，循环操作可通过迭代器进行遍历。这里，迭代器定义为一种接口，并包含了两个方法，即 hasNext 和 next 方法，并可遍历集合、范围、字符串，以及其他可表示为元素序列的实体。



当执行循环访问操作时，须提供 iterator()方法。由于字符串并未定义该方法，因而在 Kotlin 中表示为扩展函数。关于扩展，第 7 章将对此加以讨论。

Kotlin 提供了 3 种循环类型，即 for、while 和 do…while，且与其他编程语言的工作方式均为相同，因而下面对其进行简要介绍。

1. for 循环

Java 中经典的 for 循环（须显式定义迭代器）并未出现于 Kotlin 中，下列示例展示了 Java 中的此类循环方式。

```
//Java
String str = "Foo Bar";
for(int i=0; i<str.length(); i++)
System.out.println(str.charAt(i));
```

当在开始和结束之间对集合各项进行循环遍历时，可简单地使用 for 循环，如下所示：

```
var array = arrayOf(1, 2, 3)

for (item in array) {
    print(item)
}
```

另外，无须使用循环体也可对其进行定义，如下所示：

```
for (item in array)
    print(item)
```

若某个集合定义为泛型集合，那么，其中的数据项将智能转换为与泛型集合类型对应的数据类型。换而言之，如果某个集合包含类型为 Int 的数据元素，则数据项会智能转换为 Int 类型，如下所示：

```
var array = arrayOf(1, 2, 3)

for (item in array)
    print(item) // item is Int
```

除此之外，还可通过索引循环遍历集合，如下所示：

```
for (i in array.indices)
    print(array[i])
```

其中，参数 array.indices 返回包含全部索引的 IntRange，这等价于(1.. array.length - 1)。另外，还存在另一个 withIndex 库方法，可返回 IndexedValue 属性列表，其中包含了一个索引和一个数值，并可通过下列方式析构数据元素：

```
for ((index, value) in array.withIndex()) {
    println("Element at $index is $value")
}
```

此处，(index, value）结构称作析构声明，第 4 章将对此加以讨论。

2. while 循环

当对应的条件表达式为 true 时，while 循环重复执行某个代码块，如下所示：

```
while (condition) {
    // code
}
```

另外，如果条件表达式返回 true，do…while 循环同样重复执行代码块，如下所示：

```
do {
    // code
} while (condition)
```

与 Java 不同，Kotlin 可使用声明于 do…while 循环中的变量，并将其用作条件，如下所示：

```
do {
    var found = false
    //...
} while (found)
```

while 和 do…while 循环之间的主要差别在于，何时计算条件表达式。其中，while 循环在代码执行前检测相关条件：如果该条件不为 true，则不执行代码。另外一方面，do…while 循环则首先执行循环体，随后计算条件表达式。因而，循环体至少执行一次。如果对应条件为 true，则循环重复执行；否则循环结束。

3. 其他循环结构

通过内建的库函数，还存在其他一些集合遍历方式，例如 forEach。第 7 章将对此加以讨论。

2.9.4　break 和 continue

Kotlin 中的全部循环结构均支持经典的 break 和 continue 语句。continue 语句将继续执行循环的下一次迭代，而 break 则终止执行最内部的闭合循环，如下所示：

```
val range = 1..6

for(i in range) {
    print("$i ")
}

// prints: 1 2 3 4 5 6
```

下面加入相关条件，并在该条件满足时退出循环，如下所示：

```
val range = 1..6

for(i in range) {
    print("$i ")

    if (i == 3)
        break
}

// prints: 1 2 3
```

当处理内嵌循环时，break 和 continue 语句十分有用，并可简化控制流，显著降低所执行的工作量，进而节省宝贵的 Android 资源。下面考察内嵌循环，并尝试退出外部循环，如下所示：

```
val intRange = 1..6
val charRange = 'A'..'B'

for(value in intRange) {
    if(value == 3)
        break

    println("Outer loop: $value ")

    for (char in charRange) {
        println("\tInner loop: $char ")
    }
}

// prints
Outer loop: 1
    Inner loop: A
    Inner loop: B
Outer loop: 2
```

```
    Inner loop: A
    Inner loop: B
```

其中，在第三次迭代时，使用 break 语句终止了外部循环，因而内嵌循环也随之结束。注意，\t 转义序列可在控制台中实现缩进格式。另外，还可使用 continue 语句略过循环中的当前迭代，如下所示：

```
val intRange = 1..5

for(value in intRange) {
    if(value == 3)
        continue

    println("Outer loop: $value ")

    for (char in charRange) {
        println("\tInner loop: $char ")
    }
}

// prints
Outer loop: 1
    Inner loop: A
    Inner loop: B
Outer loop: 2
    Inner loop: A
    Inner loop: B
Outer loop: 4
    Inner loop: A
    Inner loop: B
Outer loop: 5
    Inner loop: A
    Inner loop: B
```

其中，若当前值等于 2，则略过外部循环中的当前迭代。

continue 和 break 语句均在闭合循环中执行相关操作。然而，某些时候，可能需要从内部另一个循环中终止或略过循环的迭代。例如，从内部循环中终止外部循环迭代，如下所示：

```
for(value in intRange) {
    for (char in charRange) {
        // How can we break outer loop here?
    }
}
```

然而，continue 和 break 语句包含了两种形式，即标记（注解）形式和非标记形式。前述内容曾对非标记形式有所讨论，下面采用标记形式处理当前问题。下列示例展示了标记 break 语句的使用方式。

```
val charRange = 'A'..'B'
val intRange = 1..6

outer@ for(value in intRange) {
    println("Outer loop: $value ")

    for (char in charRange) {
        if(char == 'B')
            break@outer

        println("\tInner loop: $char ")
    }
}

// prints
Outer loop: 1
    Inner loop: A
```

其中，@outer 表示为标记名。根据惯例，标记名以@开始，随后是标记名。另外，标记置于循环之前。对循环进行标记支持使用合法的 break（break@outer），这也是一种终止循环执行（由该标记所引用）的方法。之前的有效 break（包含标记的 break）将跳跃至（采用该标记标识的）循环之后的执行点处。

设置 return 语句将退出全部循环，并从匿名或命名函数中返回，如下所示：

```
fun doSth() {
    val charRange = 'A'..'B'
    val intRange = 1..6

    for(value in intRange) {
        println("Outer loop: $value ")

        for (char in charRange) {
            println("\tInner loop: $char ")
            return
        }
    }
}
```

```
// usage
println("Before method call")
doSth()
println("After method call")

// prints
Outer loop: 1
    Inner loop: A
```

调用该方法后，将显示下列输出结果：

```
Outer loop: 1
    Inner loop: A
```

2.10　异　　常

大多数 Java 程序设计书籍中均提出了验证检查这一概念，例如 *Effective Java* 一书。这也表明，通常需要对参数或对象的状态进行验证，如果验证检测无效，则抛出异常。Java 异常机制包含两种类型的异常，即检查型异常和非检查型异常。

非检查型异常表示，开发人员不必强制使用 try…catch 代码块捕捉异常。默认状态下，异常会上行至调用栈，因而可确定是否对其进行捕捉。如果忘记了捕捉异常，则该异常将会上行至调用栈，终止线程的执行并显示相应消息（并以此提醒用户），如图 2.10 所示。

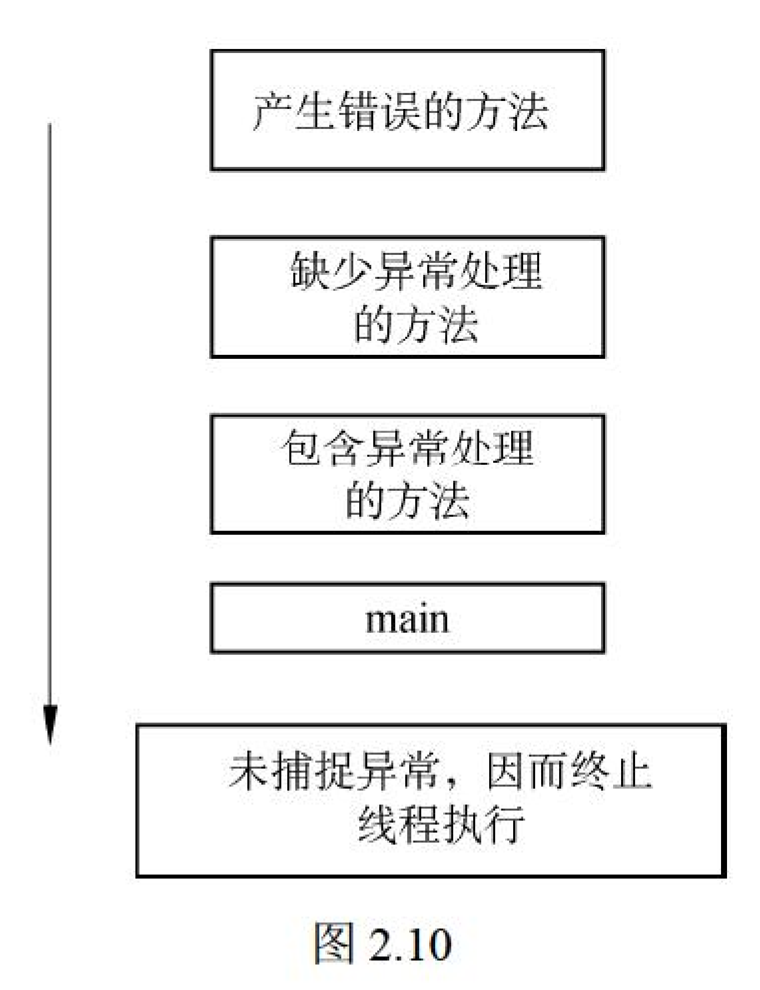


图 2.10

Java 内建了强大的异常机制，多数时候会强制开发人员显式地标注可能会抛出异常的函数，同时显式地通过 try…catch 代码块捕捉各个异常（检查型异常）。这对于小型项目来说工作良好，但在大型应用程序中，这通常会产生冗余代码，如下所示：

```
// Java
try {
    doSomething()
} catch (IOException e) {
    // Must be safe
}
```

如果未将异常传递至调用栈，则可通过空 catch 块对其进行忽略。因此，异常将无法被正常处理并于随后消失。此类代码会掩盖某些较为重要的问题，同时会带来安全问题。

在讨论 Kotlin 的异常处理机制之前，下面首先比较一下两种类型的异常，如表 2.4 所示。

表 2.4

代码	检查型异常	非检查型异常
函数声明	需要通过函数确定所抛出的异常内容	函数声明不包含与抛出异常相关的信息
异常处理	抛出异常的函数应位于 try…catch 代码块中	可捕捉异常，并根据需要执行相关操作，但这并非是强制要求。异常将上行至调用栈

Kotlin 和 Java 异常机制之间最大的差异是，在 Kotlin 中，全部异常均为非检查型，这也意味着，无须利用 try…catch 包围某个方法，即使这是一个 Java 方法且有可能抛出被捕捉的异常。如下所示：

```
fun foo() {
    throw IOException()
}

fun bar() {
    foo () // no need to surround method with try-catch block
}
```

该方案解决了代码冗余问题，并改善了安全性，即无须引入空的 catch 块。

Kotlin 的 try…catch 代码块等同于 Java 的 try…catch 代码块，如下所示：

```
fun sendFormData(user: User?, data: Data?) { // 1
    user ?: throw NullPointerException("User cannot be null")
    // 2
```

```
    data ?: throw NullPointerException("Data cannot be null")
    //do something
}

fun onSendDataClicked() {
    try { // 3
        sendFormData(user, data)
    } catch (e: AssertionError) { // 4
        // handle error
    } finally { // 5
        // optional finally block
    }
}
```

- 在注释 1 中，异常未在函数签名中设置，这一点与 Java 类似。
- 在注释 2 中，检查了数据的有效性，并抛出 NullPointerException（注意，当创建新的对象实例时，无须使用 new 关键字）。
- 在注释 3 中，try…catch 代码块类似于 Java 中的对应结构。
- 在注释 4 中，仅处理特定异常（即 AssertionError 异常）。
- 在注释 5 中，总是执行 finally 代码块。

相应地，可能会忽略 0 或多个 catch 和 finally 代码块，但至少应设置一个 catch 或 finally 代码块。

在 Kotlin 中，异常处理 try 定义为一个表达式，因而可返回一个值，并将其赋予某个变量中。实际赋值结果通常是所执行的代码块的最后一个表达式。下列示例显示了特定的 Android 应用程序是否已经安装在当前设备中。

```
val result = try { // 1
    context.packageManager.getPackageInfo("com.text.app", 0) // 2
    true
} catch (ex: PackageManager.NameNotFoundException) { // 3
    false
}
```

- try…catch 代码块表示为独立表达式函数所返回的返回值。
- 如应用程序已安装完毕，getPackageInfo 将返回一个值（该值被忽略），包含 true 表达式的下一行代码将被执行。这也是 try 代码块所执行的最后一项操作，因而其值将被赋予至某个变量中（true）。

若应用程序尚未安装，getPackageInfo 将抛出 PackageManager.NameNotFoundException，并执行 catch 代码块。其中，catch 代码块的最后一行包含了 false 表达式，因而其值将被赋予

至某个变量中。

2.11　编译期常量

由于 val 变量具有只读性质，因而在大多数时候，可将其视作常量。需要注意的是，其初始化过程可能会被延迟。这也表明，某些时候，val 变量可能不会在编译期内被初始化。例如，将当前方法的结果值赋予某个变量，如下所示：

```
val fruit:String = getName()
```

该值将在运行期内被赋值。因此，有时需要了解编译期内该值的具体内容。当需要将参数传递至注解中时，方需要使用到实际值。这里，注解由注解处理器进行操作，并在应用程序启动之前即开始运行，如图 2.11 所示。

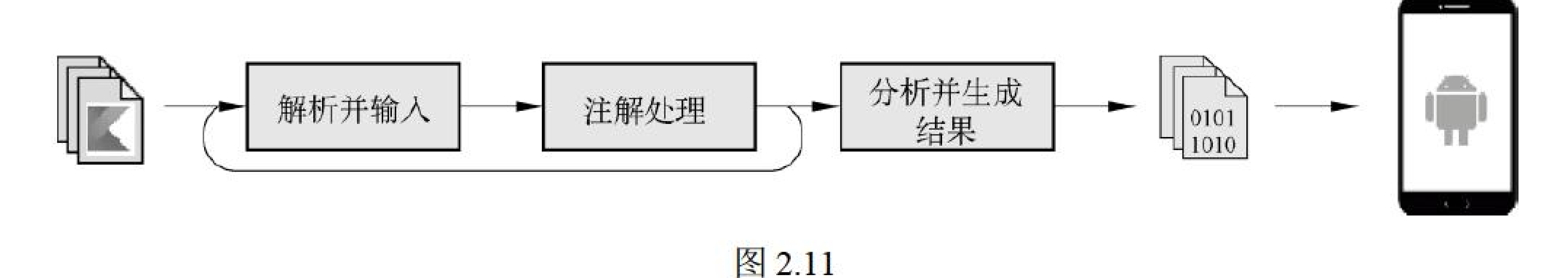


图 2.11

为了确保对应值在编译期已知（进而可被注解处理器予以处理），需要利用 const 修饰符对其进行标记。下面定义一个注解类 MyLogger，并利用单一参数定义最大日志项，并以此对 Test 类进行注解，如下所示：

```
const val MAX LOG ENTRIES = 100
    @MyLogger(MAX LOG ENTRIES )
    // value available at compile time
    class Test {}
```

关于 const 的使用，存在一些限制条件，读者须对此有所了解。首先，须采用基本类型或 String 类型进行初始化。其次，须在代码最上方或作为对象成员予以声明，第 4 章将对此加以讨论。最后，对应内容无法包含自定义 getter 方法。

2.12　委 托 机 制

Kotlin 针对委托机制提供了强有力的支持，与 Java 相比，这一点得到了明显的改善。若情况属实，Android 开发过程中可实现大量的委托应用程序，第 8 章将对此予以解释。

2.13 本章小结

本章讨论了变量、值以及常量之间的差别，同时还涉及了 Kotlin 中的基本数据类型（包括范围）。除此之外，本章还介绍了 Kotlin 的类型机制（严格的空安全检测），以及可空引用的处理方式（通过各种操作符和智能转换）。通过类型引用和各种控制结构（Kotlin 将其视为表达式），可编写更加简洁的代码。最后，本章还考察了异常的处理方式。

第 3 章将讨论函数及其不同的定义方式，同时还将探讨单一表达式函数、默认参数、命名参数语法以及各种修饰符。

第 3 章　函　　数

第 2 章介绍了 Kotlin 变量、类型机制以及控制结构。当构建应用程序时，还须使用到构造块以定义各种结构。在 Java 中，类表示为代码的构造块；另外一方面，Kotlin 支持函数编程，因而可在不使用类的情况下构建程序和库。在 Kotlin 中，函数是最基本的构造块，本章主要介绍 Kotlin 中的函数，以及各种函数特性和类型。

本章主要涉及以下内容：

- Kotlin 中基本的函数应用。
- Unit 返回类型。
- vararg 参数。
- 单一表达式函数。
- 尾递归函数。
- 默认参数值。
- 命名参数值。
- 顶级函数。
- 本地函数。
- Nothing 返回类型。

3.1　基本的函数声明和应用

"Hello World!"是测试某种语言的最为常见的程序，作为一个完整程序，对应结果将在控制台中显示"Hello World!"。同样，本节也将从该程序开始。Kotlin 程序基于某个函数且仅基于函数——无须使用类。因此，Kotlin 中的"Hello World!"程序如下所示：

```
// SomeFile.kt
fun main(args: Array<String>) { // 1
    println("Hello, World!")    // 2, Prints: Hello, World!
}
```

- 函数定义了单一参数 args，其中包含了运行当前程序的参数数组（源自命令行）。同时，参数定义为非空类型，当程序启动且不包含任何参数时，空数组将传递至方法中。

- println 函数则是定义于 Kotlin 标准库中的函数，等价于 Java 函数中的 System.out.println。

该程序涵盖了 Kotlin 中的诸多特征，例如函数的外观，且无须使用任何类即可定义函数。下面首先分析函数的结构。该函数始于关键字 fun，随后是函数名以及括号中的参数以及函数体。下列简单函数将返回一个值：

```
fun double(i: Int): Int {
    return 2 * i
}
```



人们常常容易混淆方法与函数之间的差异。一般定义如下所示：

函数表示为一个代码段，并通过名称被调用。方法则是与类实例（对象）相关联的函数，有时也称作成员函数。

简单地讲，类中的函数称作方法。在 Java 中，官方仅支持方法一说，但在学术环境中，静态方法实际上即是函数。在 Kotlin 中，函数定义为不与任何对象关联。

在 Kotlin 中，函数调用的语法与 Java 以及大多数现代程序设计语言相同，如下所示：

```
val a = double(5)
```

代码调用了 double 函数，并将其返回值赋予某个变量中。下面考察 Kotlin 函数的参数和返回类型。

3.1.1　参数

Kotlin 函数中的参数采用 Pascal 标记进行声明，各个参数的类型须显式确定。另外，全部参数均定义为只读变量。同时，不存在相关方法可对参数进行修改，此类行为将引发错误操作。在 Java 语言中，程序员常对此加以滥用。如果确有此类需求，可显式地对参数予以保护，即利用相同的名字声明局部变量，如下所示：

```
fun findDuplicates(list: List<Int>): Set<Int> {
    var list = list.sorted()
    //...
}
```

上述代码虽然可正常工作，但却是一种较差的操作行为，同时还会显示一条警告消息。一种较好的方法是根据提供的数据以及功能命名参数和变量。随后，对应名称应在大多数时候具有唯一性。



在编程社区内，实参和形参通常被认为是一种事物。但二者并不可交换使用，其原因在于：二者具有不同的含义。其中，实参表示函数调用时传递至参数的实际值；形参则表示函数声明内部所声明的变量。考察下列示例：

```
fun printSum(a1: Int, a2: Int) { // 1.
    print(a1 + a2)
}
add(3, 5) // 2.

1 - a1 and a2 are parameters
2 - 3 and 5 are arguments
```

类似于 Java，Kotlin 中的函数可包含多个参数，如下所示：

```
fun printSum(a: Int, b: Int) {
    val sum = a + b
    print(sum)
}
```

函数中参数可以是参数声明类型的子类型。如前所述，在 Kotlin 中，所有非空类型的超类型为 Any，因而在接受全部类型时，即可使用 Any，如下所示：

```
fun presentGently(v: Any) {
    println("Hello. I would like to present you: $v")
}

presentGently("Duck")
// Hello. I would like to present you: Duck
presentGently(42)
// Hello. I would like to present you: 42
```

对于参数为 null 这一类情形，类型须定义为可空类型。需要注意的是，Any?表示为全部可空以及非空类型的子类型，因而可传递任意类型的对象作为参数，如下所示：

```
fun presentGently(v: Any?) {
    println("Hello. I would like to present you: $v")
}

presentGently(null)
// Prints: Hello. I would like to present you: null
presentGently(1)
// Prints: Hello. I would like to present you: 1
presentGently("Str")
// Prints: Hello. I would like to present you: Str
```

3.1.2　返回函数

截止到目前，大多数函数均采用类似于过程的方式加以定义（不返回任何值的函数）。实际上，Kotlin 中不包含任何过程，全部函数均返回某个值。若未加指定，默认返回值为 Unit 实例。出于演示目的，下面对其进行显式设置：

```
fun printSum(a: Int, b: Int): Unit { // 1
    val sum = a + b
    print(sum)
}
```

与 Java 不同，Kotlin 可在函数名以及参数之后定义返回类型。

Unit 对象等价于 Java 中的 void，但也可视为任意其他对象，因而可将其存储于变量中，如下所示：

```
val p = printSum(1, 2)
println(p is Unit) // Prints: true
```

当然，Kotlin 编码规则指出，当函数返回 Unit 时，那么，类型定义应被忽略。通过这一方式，代码变得更具可读性且易于理解，如下所示：

```
fun printSum(a: Int, b: Int) {
    val sum = a + b
    print(sum)
}
```



Unit 定义为单例对象，也就是说，仅存在单一 Unit 实例。因此，下列 3 个条件均为 true：

```
println(p is Unit) // Print: true
println(p == Unit) // Print: true
println(p === Unit) // Print: true
```

Kotlin 支持单例模式，第 4 章将对此加以讨论。

当采用 Unit 返回类型从函数中返回输出结果时，可简单地使用不包含任何值的 return 语句，如下所示：

```
fun printSum(a: Int, b: Int) { // 1
    if(a < 0 || b < 0) {
        return // 2
    }
    val sum = a + b
    print(sum)
    // 3
}
```

- 在注释 1 中，未设定任何返回类型，因而返回类型隐式地设置为 Unit。
- 在注释 2 中，可使用不包含任何值的 return 语句。
- 当函数返回 Unit 时，return 调用为可选项。

除此之外，还可返回 Unit，但这通常会导致误解且缺乏可读性，因而不建议使用。

当制定返回类型时，除了 Unit 之外，通常还会显式地返回值，如下所示：

```
fun sumPositive(a: Int, b: Int): Int {
    if(a > 0 && b > 0) {
        return a + b
    }
    // Error, 1
}
```

在注释 1 中，函数并不会对其进行编译，其原因在于：未指定返回值，且 if 条件未被满足。

通过添加第二条 return 语句，即可解决当前问题，如下所示：

```
fun sumPositive(a: Int, b: Int): Int {
    if(a >= 0 && b >= 0) {
        return a + b
    }
    return 0
}
```

3.2　vararg 参数

某些时候，参数的数量事先未知。此时，可向参数添加一个 vararg 修饰符，进而使函数可接收任意数量的参数。例如，下列函数将输出多个整数之和。

```
fun printSum(vararg numbers: Int) {
    val sum = numbers.sum()
    print(sum)
}

printSum(1,2,3,4,5) // Prints: 15
printSum()          // Prints: 0
```

参数可在方法内部作为加载全部数值的数组予以访问。其中，数组类型对应于 vararg 参数类型。正常状态下，期望为加载特定类型的泛型数组（Array<T>）；然而，Kotlin 针对

Int 数组定义了优化类型，即 IntArray，因而应使用该类型的数组。下列示例表示为包含 String 类型的 vararg 参数。

```
fun printAll(vararg texts: String) {
// Inferred type of texts is Array<String>
     val allTexts = texts.joinToString(",")
     println("Texts are $allTexts")
}

printAll("A", "B", "C") // Prints: Texts are A,B,C
```

注意，可在 vararg 参数前、后确定多个参数，且参数之间需要一一对应，如下所示：

```
fun printAll(prefix: String, postfix: String, vararg texts: String)
{
    val allTexts = texts.joinToString(", ")
    println("$prefix$allTexts$postfix")
}

printAll("All texts: ", "!") // Prints: All texts: !
printAll("All texts: ","!" , "Hello", "World")
// Prints: All texts: Hello, World!
```

除此之外，提供与 vararg 形参的实际参数应为特定类型的子类型，如下所示：

```
fun printAll(vararg texts: Any) {
    val allTexts = texts.joinToString(",") // 1
    println(allTexts)
}

// Usage
printAll("A", 1, 'c') // Prints: A,1,c
```

在注释 1 中，joinToString 可在列表上被调用，并将各元素连接为一个独立的字符串。在第一个参数上，定义了一个分隔符。

在每个函数声明中，仅可存在一个 vararg 参数，这也是 vararg 应用时的一个限制条件。

当调用 vararg 参数时，可逐一传递参数值，但也可传递值数组。这可通过 spread 操作符实现（*前缀数组），如下所示：

```
val texts = arrayOf("B", "C", "D")
printAll(*texts) // Prints: Texts are: B,C,D
printAll("A", *texts, "E") // Prints: Texts are: A,B,C,D,E
```

3.3　单表达式函数

在典型的程序设计过程中，许多函数仅包含一个表达式，如下所示：

```
fun square(x: Int): Int {
    return x * x
}
```

另一个例子则常见于 Android 项目中，并采用了 Activity 中的一种模式，相关方法从视图中获取文本，或者从视图中提供其他数据以供用户使用，如下所示：

```
fun getEmail(): String {
    return emailView.text.toString()
}
```

上述两个函数均返回单表达式。在第一个示例中，返回值表示为 x * x 乘法运算结果；在第二个示例中，返回结果为 emailView.text.toString()。这一类函数常用于 Android 项目中，其中包括：

- 获取某些小型操作（例如之前的 square 函数）。
- 使用多态向某个特定类提供数值。
- 仅用于生成某个对象的函数。
- 在体系结构层之间传递数据的函数（例如，Activity 在视图和用户之前传递数据）。
- 基于递归的函数式编程风格函数。

此类函数经常会使用到，因而针对此类函数设置了对应标记。当函数返回单表达式时，则可忽略花括号和函数体。采用这一方式定义的函数称作单表达式函数。下面更新 square 函数，并将其定义为单表达式函数，如图 3.1 所示。

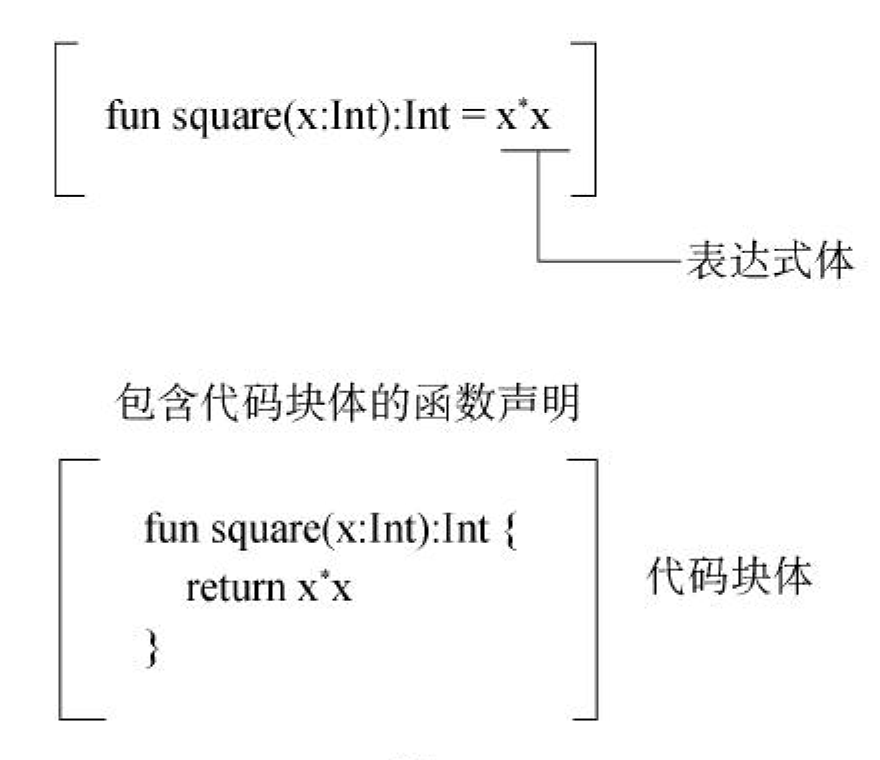


图 3.1

不难发现，单表达式函数包含表达式体，而非代码块体。该符号更为短小，但整体仅为一个单表达式。

在单表达式函数中，声明返回类型则视为可选项——编译器可通过表达式类型进行推断，进而简化了 square。对应的定义方式如下所示：

```
fun square(x: Int) = x * x
```

在 Android 项目中，单表达式函数十分常见。下面考察 RecyclerView 适配器，用于提供布局 ID 并创建 ViewHolder，如下所示：

```
class AddressAdapter : ItemAdapter<AddressAdapter.ViewHolder>() {
    override fun getLayoutId() = R.layout.choose address view
    override fun onCreateViewHolder(itemView: View) =
    ViewHolder(itemView)

    // Rest of methods
}
```

在下面的示例中，在单表达式函数的基础上，实现了具有较好可读性的编码内容。另外，单表达式函数在函数式编程中也十分常见，稍后将讨论具体示例（即尾递归函数）。同时，单表达式函数还可与 when 结构协同使用。下列代码实现了这种连接方式，并根据键值从对象中获取特定数据。

```
fun valueFromBooking(key: String, booking: Booking?) = when(key) {
    // 1
    "patient.nin" -> booking?.patient?.nin
    "patient.email" -> booking?.patient?.email
    "patient.phone" -> booking?.patient?.phone
    "comment" -> booking?.comment
    else -> null
}
```

此处并不需要定义某个类型，对应类型可从 when 表达式中进行推断。

另一个常见的 Android 示例则是结合使用 when 表达式以及 activity onOptionsItem Selected，进而处理上方工具栏的单击操作，如下所示：

```
override fun onOptionsItemSelected(item: MenuItem): Boolean = when
{
    item.itemId == android.R.id.home -> {
        onBackPressed()
        true
    }
    else -> super.onOptionsItemSelected(item)
}
```

另一个体现单表达式函数作用的示例则是，在独立对象上链接多项操作，如下所示：

```
fun textFormatted(text: String, name: String) = text
                  .trim()
                  .capitalize()
                  .replace("{name}", name)
val formatted = textFormatted("hello, {name}", "Marcin")
println(formatted) // Hello, Marcin
```

不难发现，单表达式函数使代码更加简洁，并改善了代码的可读性。单表达式函数常用于 Kotlin Android 项目中，并且在函数式编程中十分常见。

命令式编程和声明式编程



命令式编程范例描述了所需的实际步骤序列，进而执行某项操作。对于大多数程序员来讲，这种方式较为直观。

声明式编程范例描述了期望结果，但无须按照各步骤予以实现（行为实现）。这也表明，程序通过表达式或声明完成，而非语句。函数式和逻辑式编程均具有显著的声明式程序设计风格。与命令式编程相比，声明式编程更加短小，且具有较好的可读性。

3.4　尾递归函数

递归函数是指实现了自身调用的函数，下面考察递归函数 getState。

```
fun getState(state: State, n: Int): State =
    if (n <= 0) state // 1
    else getState(nextState(state), n - 1)
```

递归是函数式编程中的重要内容，但每次递归调用需要在栈中保存上一个函数的返回地址。当应用程序深度递归时（栈中存在大量的函数），将会抛出 StackOverflowError。这也是递归应用中较为严重的一个问题。

针对这一问题，经典的解决方案是使用迭代，而非递归，但前者的表达性较差，如下所示：

```
fun getState(state: State, n: Int): State {
    var state = state
    for (i in 1..n) {
        state = state.nextState()
    }
    return state
}
```

对此，一种较好方法是尾递归函数，诸如 Kotlin 等现代编程语言均对此予以支持。尾递归函数是一种特殊的递归函数，其中，函数在执行最后一次操作时调用自身（换而言之，递归发生在函数的最后一次操作）。这可通过编译器优化递归调用，并以更加高效的方式执行递归操作，而无须担心潜在的 StackOverflowError。当实现函数尾递归时，须使用 tailrec 修饰符对其进行标记，如下所示：

```
tailrec fun getState(state: State, n: Int): State =
    if (n <= 0) state
    else getState(state.nextState(), n - 1)
```

为了进一步检测尾递归的工作方式，可对代码进行编译、反编译操作，如下所示（简化后的代码）：

```
public static final State getState(@NotNull State state, int n)
{
    while(true) {
        if(n <= 0) {
            return state;
        }
        state = state.nextState();
        n = n - 1;
    }
}
```

上述实现基于迭代操作，因而不存在栈上溢问题。为了使 tailrec 正常工作，须满足某些需求条件，其中包括：

- 函数须在执行最后一次操作时调用自身。
- 不可在 try/catch/finally 块中使用。
- 在本书编写时，仅支持将 Kotlin 编译为 JVM。

3.5　调用函数的不同方式

在某些场合下，需要调用某个函数，并通过所选参数。在 Java 中，可创建同一方法的多个重载方法，但该方案包含某些局限性。第一个问题是，既定方法的排列数量增长十分迅速（2^n），进而难以实现有效的管理。第二个问题是，重载方法之间彼此难以区分。编译器可能知晓调用哪一个重载方法；但当某个方法定义了包含相同类型的多个参数时，则无法进一步定义相应的重载方法。这也是为什么常常需要向某个方法中传递多个 null 值的原因，如下所示：

```
// Java
printValue("abc", null, null, "!");
```

上述方法极大地降低了代码的可读性。在 Kotlin 中，此类问题不复存在。Kotlin 包含了默认参数和命名参数语法这一类特征。

3.5.1　默认参数值

默认参数在 C++中体现的较为明显，一些较为古老的编程语言也对此予以支持。默认参数向参数提供了一个值，以防在方法调用其间参数缺失。每个函数参数均包含了一个默认值，可能是与特定类型相匹配的任意值，包括 null。据此，可简单地定义函数，并可通过多种方式对其加以调用。下列代码显示了包含默认值的函数示例。

```
fun printValue(value: String, inBracket: Boolean = true,
               prefix: String = "", suffix: String = "") {
    print(prefix)
    if (inBracket) {
        print("(${value})")
    } else {
        print(value)
    }
    println(suffix)
}
```

通过向各个参数提供数值，该函数的使用方式与常规函数（未包含默认值的函数）并无两样，如下所示：

```
printValue("str", true, "","") // Prints: (str)
```

对于默认参数值，可仅向不包含默认值的参数提供数据，进而调用某个函数，如下所示：

```
printValue("str") // Prints: (str)
```

相应地，还可提供没有默认值的全部参数，或者只有某些参数包含默认值，如下所示：

```
printValue("str", false) // Prints: str
```

3.5.2　命名参数语法

有时，仅须向最后一个参数传递数值。假设针对后缀定义一个值，而非前缀和 inBracket（已在前缀之前加以定义）。正常情况下，须针对之前的所有参数提供相应值，包括默认参数值，如下所示：

```
printValue("str", true, true, "!") // Prints: (str)
```

通过命名参数语法，可通过参数名传递特定的参数，如下所示：

```
printValue("str", suffix = "!") // Prints: (str)!
```

这将支持更为灵活的语法形式。当调用某个函数时，仅提供所选参数即可。考虑到代码的可读性，该形式常用于确定参数的真正含义，如下所示：

```
printValue("str", inBracket = true) // Prints: (str)
printValue("str", prefix = "Value is ") // Prints: Value is str
printValue("str", prefix = "Value is ", suffix = "!! ")
// Prints: Value is str!!
```

只要提供了不包含默认值的全部参数，即可采用命名参数语法并以任意顺序设置所需参数。另外，参数的顺序具有一定的相关性，如下所示：

```
printValue("str", inBracket= true, prefix = "Value is ")
// Prints: Value is (str)
printValue("str", prefix = "Value is ", inBracket= true)
// Prints: Value is (str)
```

其中，参数的顺序有所不同，但两条语句的调用彼此等价。

另外，还可将命名参数语法与常规调用结合使用。唯一的限制条件是：如果使用命名语法，则无法针对下一个参数再次使用，如下所示：

```
printValue ("str", true, "")
printValue ("str", true, prefix = "")
printValue ("str", inBracket = true, prefix = "")
printValue ("str", inBracket = true, "") // Error
printValue ("str", inBracket = true, prefix = "", "") // Error
```

该特性可通过更加灵活的方式调用相关方法，且无须定义多个重载方法。

对于 Kotlin 参数，命名参数语法还具有额外的特征。需要注意的是，当改变某个参数名时，由于该名称可能会用于其他项目中，因而将会产生错误。Android Studio 对此十分关注，如果通过内建重构工具对参数名称进行重命名，该操作仅可工作于当前项目中。当使用命名参数语法时，Kotlin 库生成器一般会对此予以谨慎处理。参数名称的改变会对 API 产生负面影响。注意，当调用 Java 函数时，命名参数语法将无法继续使用——Java 字节码不会保留函数参数名称。

3.6　顶级函数

通过观察可知，在“Hello, World!”程序中，main 函数未设置于任何类中。第 2 章曾

提到，Kotlin 可在顶级（top-level）位置处定义各种实体。相应地，定义于顶级位置处的函数称作顶级函数，如下所示：

```
// Test.kt
package com.example

fun printTwo() {
    print(2)
}
```

顶级函数可用于代码的任何地方，其调用方式与局部函数相同。当访问顶级函数时，须通过 import 语句显式地将其导入至文件中。在 Android Studio 中，函数将显示于提示列表中，当函数被选择（使用）后，将自动添加导入。下列代码在 Test.kt 中定义了顶级函数，并在 Main.kt 文件中对其加以使用。

```
// Test.kt
package com.example

fun printTwo() {
    print(2)
}

// Main.kt
import com.example.printTwo

fun main(args: Array<String>) {
    printTwo()
}
```

尽管顶级函数十分有用，但仍须对其予以理智使用。定义公有型顶级函数将会增加代码提示列表中函数的数量（这里，提示列表是指，当编写代码时，IDE 所建议的提示方法列表）。如果顶级函数名称未予清晰定义，则会与本地对应函数相混淆。以下内容是与顶级函数相关的一些较好的示例：

- factorial。
- maxOf 和 minOf。
- listOf。
- println。

而下列函数则是较差的顶级函数示例：

- sendUserData。
- showPossiblePlayers。

这一规则仅适用于 Kotlin 面向对象程序设计项目。在面向函数的程序设计项目中，此类名称均为有效的顶级函数名称。但是，此处假设几乎所有的函数均在顶层定义的，而不是在方法中定义的。

通常，函数可定义在特定的模块或特定类中。为了限制函数的可见性（可用范围），可使用相应的可见性修饰符，第 4 章将对此加以讨论。

3.7　顶级函数的底层机制

对于 Android 项目，Kotlin 编译为 Java 字节码，并运行于 Dalvik 虚拟机（Android 5.0 之前）或 Android 运行期（Android 5.0 之后的版本），两种虚拟机仅执行定义于类中的代码。针对这一问题，Kotlin 编译器针对顶级函数生成类。相应地，类名根据文件名以及 Kt 后缀构造。在这样的类中，全部函数和属性均为静态。例如，假设在 Printer.kt 文件中定义了下列函数：

```
// Printer.kt
fun printTwo() {
    print(2)
}
```

Kotlin 代码将编译为 Java 字节码。其中，所生成的字节码类似于下列 Java 类中生成的代码：

```
// Java
public final class PrinterKt { // 1
    public static void printTwo() { // 2
        System.out.print(2); // 3
    }
}
```

- 对注释 1，PrinterKt 表示为基于文件名和 Kt 后缀生成的名称。
- 对注释 2，全部顶级函数和属性均编译为静态方法和变量。
- 对注释 3，print 定义为 Kotlin 函数，但作为内联函数，在编译期内，其调用被函数体所替代。其中，函数体仅包含 System.out.println 调用。

第 5 章将对内联函数加以讨论。

在 Java 字节码级别，Kotlin 将包含更多数据（例如参数名）。另外，利用类名添加函数调用前缀，还可从 Java 文件中访问 Kotlin 顶级函数，如下所示：

```
// Java file, call inside some method
PrinterKt.printTwo()
```

通过这种方式，可全方位支持 Java 中的 Kotlin 顶级函数调用。不难发现，Kotlin 与 Java 之间可实现彼此协作。为了在 Java 中更加方便地使用 Kotlin 顶级函数，可添加注解（annotation），进而修改 JVM 生成类的名称。当使用顶级 Kotlin 属性以及 Java 类中的函数时，这将十分方便。对应注解如下所示：

```
@file:JvmName("Printer")
```

在文件上方，须添加 JvmName 注解（在包名之前）。当采取这一方式时，生成类的名称将调整为 Printer。通过将 Printer 用作类名，即可在 Java 中调用 printTwo 函数，如下所示：

```
// Java
Printer.printTwo()
```

有些时候，顶级函数需要在独立文件中加以定义，但是我们也希望它们在编译到 JVM 之后出现在同一个类中。对此，可在文件上方使用下列注解：

```
@file:JvmMultifileClass
```

例如，假设希望利用数学帮助函数（源自 Java）创建库，可定义下列文件：

```
// Max.kt
@file:JvmName("Math")
@file:JvmMultifileClass
package com.example.math

fun max(n1: Int, n2: Int): Int = if(n1 > n2) n1 else n2

// Min.kt
@file:JvmName("Math")
@file:JvmMultifileClass
package com.example.math

fun min(n1: Int, n2: Int): Int = if(n1 < n2) n1 else n2
```

随后，可通过下列方式从 Java 类中对其加以使用：

```
Math.min(1, 2)
Math.max(1, 2)
```

据此，可极大地简化文件，同时保持了易用性（从 Java 中）。

当在 Kotlin 中创建库（可直接用于 Java 类中）时，用于修改生成类名的 JvmName 注解十分有用，同时还有助于解决名称冲突问题。当创建 X.kt 文件（包含某些顶级函数或属性）以及同一个包中的 XKt 类时，即会出现命名冲突问题。根据现有规则，类名不应包含 Kt 后缀，因而这一情况基本不会出现。

3.8 局部函数

Kotlin 支持多种环境下的函数定义，例如顶级函数、成员函数（位于类、接口等内部），以及其他函数内部中的函数（局部函数）。考察下列局部函数定义：

```
fun printTwoThreeTimes() {
    fun printThree() { // 1
        print(3)
    }
    printThree() // 2
    printThree() // 2
}
```

- 对于注释 1，printThree 表示为局部函数，该函数位于另一个函数内部。
- 对于注释 2，局部函数无法从所声明的函数外部进行访问。

在局部函数中可访问的元素不需要从封闭函数中作为参数传递，因为可对其直接进行访问。例如：

```
fun loadUsers(ids: List<Int>) {
    var downloaded: List<User> = emptyList()

    fun printLog(comment: String) {
        Log.i("loadUsers (with ids $ids): $comment\nDownloaded:
                                          $downloaded") // 1
    }
    for(id in ids) {
        printLog("Start downloading for id $id")
        downloaded += loadUser(id)
        printLog("Finished downloading for id $id")
    }
}
```

对于注释 1，局部函数可访问 comment 参数和局部变量（downloaded 和 ID），此类数据定义在封闭函数内部。

如果将 printLog 定义为顶级函数，随后须作为参数传递 ids 和 downloaded，如下所示：

```
fun loadUsers(ids: List<Int>) {
    var downloaded: List<User> = emptyList()

    for(id in ids) {
        printLog("Start downloading for id $id", downloaded, ids)
```

```
        downloaded += loadUser(id)
        printLog("Finished downloading for
                 id $id", downloaded, ids))
    }
}

fun printLog(state: String, downloaded: List<User>, ids: List<Int>)
{
    Log.i("loadUsers (with ids $ids):
    $state\nDownloaded: downloaded")
}
```

上述实现不仅冗长而且难于维护。printLog 中的变化要求使用不同的参数，而参数变化则需要改变函数调用中的参数。另外，如果调整 printLog 中的 loadUsers 参数类型，同时还应修改 printLog 中的参数。对此，如果 printLog 定义为局部参数，则不会产生任何问题。这也解释了局部参数的应用时机：获取仅供单一函数使用的功能项，且对应功能使用了该函数中的数据元素（变量、值以及参数）。此外，局部函数还可调整局部变量，如下所示：

```
fun makeStudentList(): List<Student> {
    var students: List<Student> = emptyList()
    fun addStudent(name: String, state: Student.State =
                 Student.State.New) {
        students += Student(name, state, courses = emptyList())
    }
    // ...
    addStudent("Adam Smith")
    addStudent("Donald Duck")
    // ...
    return students
}
```

通过这种方式，可获取并复用 Java 中无法得到的某些功能。这也体现了局部函数的优势：某些时候，局部函数可以实现其他方式难以完成的代码析取工作。

3.9　无返回类型

有时需要定义一个仅抛出异常（非正常结束）的函数，具体如下：

● 函数可简化错误抛出机制。在某些库中，错误机制十分重要，同时需要对其提供详细的信息（具体示例参见本节中的 throwError 函数）。

- 在单元测试中，某些函数用于抛出错误。当在代码中测试错误处理过程时，这将十分有用。

针对上述各种情形，存在一个较为特殊的类，即 Nothing。Nothing 类定义为空类型，且不存在任何实例。包含 Nothing 返回类型的函数不会返回任何内容，且不会执行到 return 语句，并仅仅抛出一个异常。也就是说，函数仅返回 Nothing，且抛出一个异常。通过这种方式，可从非终止函数中（返回 Nothing）区分某些非返回值函数（例如 Java 中的 void，以及 Kotlin 中的 Unit）。下面考察此类函数示例，进而可用于简化单元测试中的错误抛出机制。

```
fun fail(): Nothing = throw Error()
```

通过使用定义环境（例如类或函数）中的数据元素，可构造复杂的错误消息机制，如下所示：

```
fun processElement(element: Element) {
    fun throwError(message: String): Nothing
    = throw ProcessingError("Error in element $element: $message")

    // ...
    if (element.kind != ElementKind.METHOD)
        throwError("Not a method")
    // ...
}
```

作为一种替代方案，这一类函数可像 throw 语句一样使用，且不会对函数返回类型产生任何影响。

```
fun getFirstCharOrFail(str: String): Char
    = if(str.isNotEmpty()) str[0] else fail()

val name: String = getName() ?: fail()

val enclosingElement = element.enclosingElement ?: throwError ("Lack of
enclosing element")
```

这可视作 Nothing 类的特点之一，就好像它是所有可能类型的子类型一样，即可空类型和非空类型。同时，这也是 Nothing 被称作空类型的原因。这意味着，不存在任何值可在运行期内包含该类型，同时它也是其他类型的子类型，如图 3.2 所示。

类型层次结构

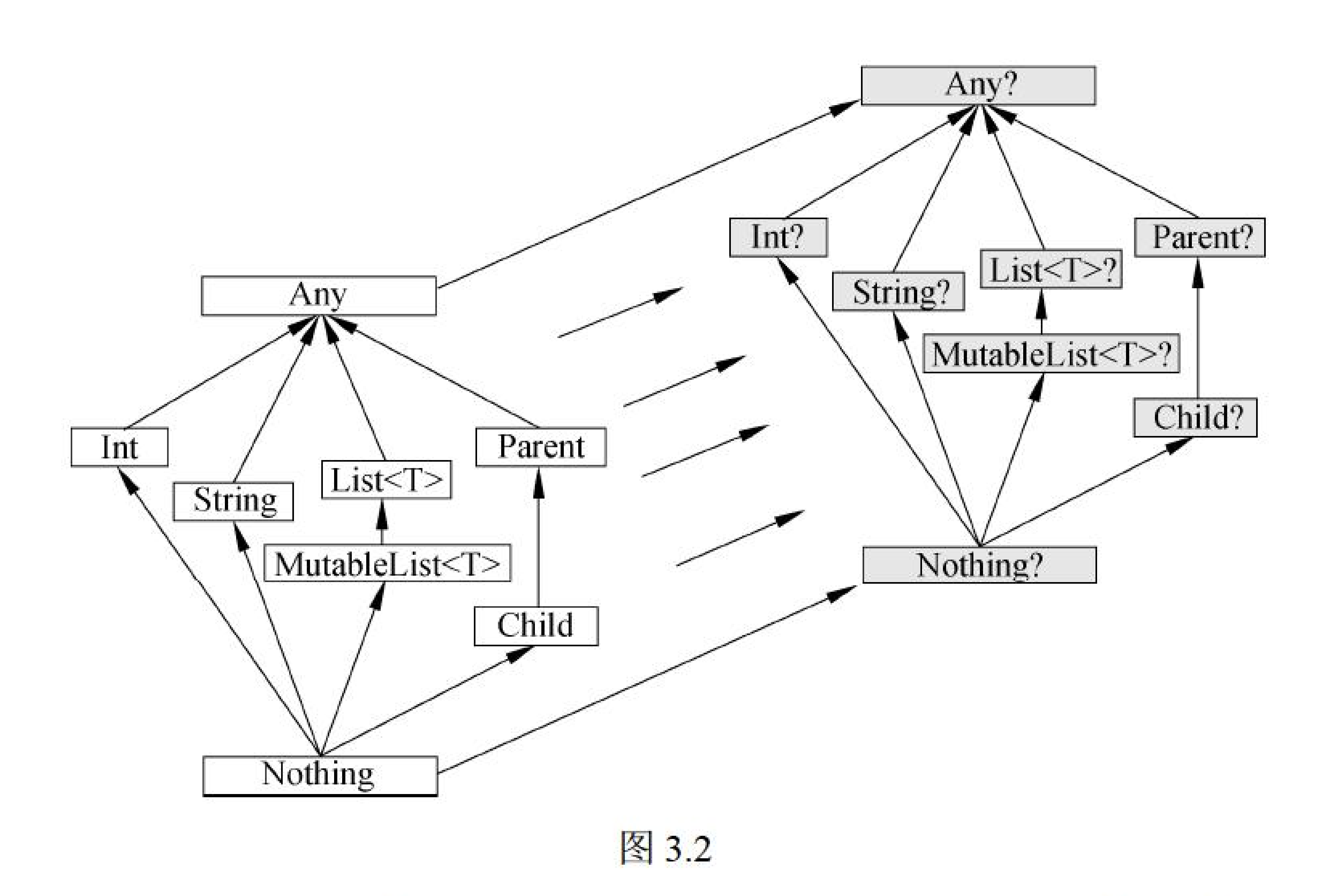


图 3.2

对于 Java 而言，空类型则是一个全新的概念，因而读者可能对此稍感陌生。实际上，这一概念十分简单。另外，无法生成 Nothing 实例，且仅存在从函数返回的错误信息，并将其作为返回类型。此外，对于 Nothing 来讲，没有必要添加任何内容以对该类型产生影响。

3.10　本 章 小 结

本章考察了函数的定义和使用方式，以及顶级函数和其他函数内部的定义方式。除此之外，本章还讨论了与函数关联的不同特征，包括 ararg 参数、默认名称以及命名参数语法。最后，本章还查看了 Kotlin 特定返回类型，包括 Unit（等价于 Java 中的 void）和 Nothing（Nothing 是一种无法定义的类型，表示为空类型且仅为异常）。

第 4 章将探讨 Kotlin 中类的定义方式。Kotlin 语言对类给予了特殊的支持，并在 Java 定义的基础上引入了大量的改进措施。

第 4 章　类 和 对 象

Kotlin 语言对 OOP 程序设计提供了完整的支持。本章将考察这一强大的结构，从而简化数据模型的定义，同时采用一种简单、灵活的方式在类结构上进行操作。此外，还将介绍诸多概念的简化和改进方式，相关概念均来自于 Java。同时，本章还将探讨不同类结构的类型、属性、初始化程序代码块以及构造方法，另外还会涉及操作符重载和接口默认实现。

本章主要涉及以下内容：

- 类声明。
- 属性。
- 属性的访问语法。
- 构造方法和初始化程序代码块。
- 构造方法。
- 继承。
- 接口。
- 数据类。
- 解构声明。
- 操作符重载。
- 对象声明。
- 对象表达式。
- 伴生对象。
- 枚举类。
- 密封类。
- 嵌套类。

4.1　类

类是 OOP 中的基本构造模块。实际上，Kotlin 类与 Java 类十分类似。然而，Kotlin 采用更为简单、简洁的语法实现了更加丰富的功能。

Kotlin 中的类采用 class 关键字加以定义。下列最简单的类声明表示名为 Person 的空类。

```
class Person
```

Person 声明未包含任何类体，但仍可通过默认的构造方法予以初始化，如下所示：

```
val person = Person()
```

即使类初始化这一类简单的任务，在 Kotlin 中也得到了简化。与 Java 不同，Kotlin 并不需要使用 new 关键字生成类实例。鉴于 Kotlin 与 Java 之间紧密的关联性，可采用相同方式实例化定义于 Java 和 Kotlin 中的类（且无须使用 new 关键字）。类实例化语法取决于生成类实例的具体语言（Kotlin 或 Java），如下所示：

```
// Instantiate Kotlin class inside Java file
Person person = new Person()
// Instantiate class inside Kotlin file
var person = Person()
```

作为一项经验法则，可在 Java 文件中使用 new 关键字；而在 Kotlin 文件中，则无须使用 new 关键字。

4.2　属　　性

属性表示为幕后字段（backing field）和访问器的组合结果，即包含 getter 和 setter 方法的幕后字段，或者是包含两种方法之一的幕后字段。另外，属性可作为顶级（直接在文件中）或成员（位于类、接口中）加以定义。

通常情况下，建议对属性加以定义（包含 getter/setter 的私有字段），而非直接访问公有字段。

Java 中私有字段的 setter 和 getter 规则



（1）getter 方法：表示为一类无参数方法，包含一个对应于属性名的名称，以及一个 get 前缀（对于 Boolean 属性，则使用 is 前缀）。

（2）setter 方法：表示为单一参数的方法，包含以 set 起始的名称，例如 setResult（String resultCode）。

Kotlin 通过语言设计确保了这一原则，该方案提供了诸多封装优点，如下所示：

- 无须改变外部 API 即可调整内部实现。
- 确保不变性（调用方法以验证对象状态）。
- 当访问成员时，可执行额外操作（例如日志操作）。

当定义顶级属性时，可简单地在 Kotlin 文件中对其进行定义，如下所示：

```
// Test.kt
val name:String
```

假设需要通过一个类存储与某人相关的基本数据，该数据可从外部 API 载入（后端），或者从本地数据库中获取。当前类应定义两个（成员）属性，即 name 和 age。下面首先考察 Java 实现，如下所示：

```
public class Person {

    private int age;
    private String name;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

该类仅包含两个属性。由于 Java IDE 可帮助我们生成访问器代码，因而无须采用手动方式编写。然而，该方案的问题是，如果 IDE 未自动生成相应的代码块，相关工作将无法进一步展开；另外，代码也变得相对冗长。开发人员将花费大量的时间阅读代码，而不是着手编写代码。阅读冗余代码将会浪费大量的时间。另外，诸如重构属性名这一类简单任务将变得更富技巧性——IDE 可能并不会更新构造方法参数名。

然而，Kotlin 可显著地降低模板代码量。对此，Kotlin 引入了内建于该语言的属性概念。下面考察前述 Java 类中的 Kotlin 等价类，如下所示：

```
class Person {
    var name: String
    var age: Int

    constructor(name: String, age: Int) {
        this.name = name
        this.age = age
    }
}
```

上述代码表示为之前 Java 类的等价内容，其中：

- constructor 方法等价于 Java 的构造方法。当生成对象实例时，该方法将被调用。
- Kotlin 编译器负责生成 getter 和 setter。

当然，可自定义 getter 和 setter 的实现内容，稍后将对此加以详细讨论。

之前全部已定义的构造函数称作次级构造函数。Kotlin 还提供了一种替代方案，并采用简洁的语法定义构造函数。对此，可将构造函数（包含全部参数）定义为类中的部分内容，此类构造函数称作主构造函数。下面将次级构造函数中的属性声明移至主构造函数中，进而适当减少代码量，如下所示：

```
class Person constructor(name: String, age: Int) {
    var name: String
    var age: Int

    init {
        this.name = name
        this.age = age
        println("Person instance created")
    }
}
```

与次级构造函数不同，在 Kotlin 中，主构造函数不可包含任何代码，因此，初始化代码须置于初始化程序代码块中（init）。在生成类时，将执行初始化程序代码块。其中，可将构造函数参数赋予其中的字段。

为了简化代码，可移除初始化程序代码块，并直接访问属性初始化程序中的构造函数参数。这可将构造函数参数直接赋予某个字段，如下所示：

```
class Person constructor(name: String, age: Int) {
    var name: String = name
    var age: Int = age
}
```

尽管代码量有所减少，但仍然涵盖了一些模板代码。其中，类型声明、属性名均重复出现（构造函数参数、字段复制以及字段自身）。若属性未包含自定义的 getter 和 setter 方法，则可直接在主构造函数中通过添加 val 或 var 对其进行定义，如下所示：

```
class Person constructor (var name: String, var age: Int)
```

最后，若主构造函数未包含任何注解（例如@Inject 等），或者可见性修饰符（例如 public, private 等），则可忽略 constructor 关键字，如下所示：

```
class Person (var name: String, var age: Int)
```

当构造函数接收某些参数时，一种较好的做法是，在新的一行中定义各个参数，进而提升代码的可读性，同时减少潜在合并冲突的概率（合并源代码存储库中的分支），如下所示：

```
class Person(
    var name: String,
    var age: Int
)
```

综上所述，在当前示例中，属性均直接定义于类主构造函数中，Kotlin 编译器为我们实现了大量工作，例如生成相应的字段和访问器（getter/setter）。

注意，此类标注法仅包含了与数据模型类相关的、最为重要的信息，包括名称、参数名、类型以及可变（val/var）信息。这也使得对应类易于阅读、理解和维护。

4.2.1　读-写属性和只读属性

在之前的示例中，全部属性均以读-写方式定义（setter 方法和 getter 方法）。当定义只读属性时，需要使用到 val 关键字，因而仅生成 getter 方法，如下所示：

```
class Person(
    var name: String,
    // Read-write property (generated getter and setter)
    val age: Int // Read-only property (generated getter)
)

// usage
val person = Person("Eva", 25)

val name = person.name
person.name = "Kate"

val age = person.age
person.age = 28 // error: read-only property
```

Kotlin 并不支持只写属性（此时仅会生成 setter 方法）。表 4.1 显示了 Kotlin 中与读、写相关的属性。

表 4.1

关键字	读	写
var	是	是
val	是	否
不支持	否	是

4.2.2 属性访问语法

Kotlin 中另一项重大改进则是属性的访问方式。在 Java 中，可通过对应方法访问属性（例如 setSpeed/getSpeed）。Kotlin 则提供了属性访问语法，并具有鲜明的表达方式。下面对两种方案加以考察，假设类 Car 中包含了单一 speed 属性，如下所示：

```
class Car (var speed: Double)

// Java access properties using method access syntax
Car car = new Car(7.4)
car.setSpeed(9.2)
Double speed = car.getSpeed();

// Kotlin access properties using property access syntax
val car: Car = Car(7.4)
car.speed = 9.2
val speed = car.speed
```

通过观察可知，在 Kotlin 中，无须添加 get、set 和括号以访问或修改对象属性。属性访问语法可直接使用++和--操作符，并与属性访问协同使用，如下所示：

```
val car = Car(7.0)
println(car.speed) // prints 7.0
car.speed++
println(car.speed) // prints 8.0
car.speed--
car.speed--
println(car.speed) // prints: 6.0
```

其中，存在两种递增（++）和递减（−−）操作符，即前递增/前递减（操作符在表达式之前定义），以及后递增/后递减（操作符在表达式之后定义），如下所示：

```
++speed //pre increment
--speed //pre decrement

speed++ //post increment
speed-- //post decrement
```

前递增和前递减操作符并不会对结果产生影响，此类操作符仅是按顺序执行。但是，当与函数调用结合使用时，情况则有所变化。

在前递增操作符中，首先获得 speed 值，递增后作为参数传递至函数中，如下所示：

```
var speed = 1.0
println(++speed) // Prints: 2.0
println(speed) // Prints: 2.0
```

在后递增操作符中，首先获得 speed 值，作为参数传递至函数中，随后递增。因此，原值将传递至函数中，如下所示：

```
var speed = 1.0
println(speed++) // Prints: 1.0
println(speed) // Prints: 2.0
```



关于前递减和后递减操作符，其工作方式也大致类似。

需要说明的是，属性访问语法不仅限于 Kotlin 中定义的类。各种方法（针对 getter 和 setter 方法，遵循 Java 规则）均可表示为 Kotlin 中的属性。

因此，可在 Java 中定义一个类，并采用属性访问语法访问 Kotlin 中的属性。下面定义一个 Fish 类，其中包含两个属性，即 size 和 isHungry。随后在 Kotlin 中初始化该类，并访问其属性，如下所示：

```
// Java class declaration
public class Fish {
    private int size;
    private boolean hungry;
    public Fish(int size, boolean isHungry) {
        this.size = size;
        this.hungry = isHungry;
    }

    public int getSize() {
        return size;
    }

    public void setSize(int size) {
        this.size = size;
```

```
    }

    public boolean isHungry() {
        return hungry;
    }

    public void setHungry(boolean hungry) {
        this.hungry = hungry;
    }
}

// Kotlin class usage
val fish = Fish(12, true)
fish.size = 7
println(fish.size) // Prints: 7
fish.isHungry = true
println(fish.isHungry) // Prints: true
```

上述代码可通过两种方式工作，因而可利用更加简洁的语法在 Kotlin 中定义 Fish 类，Kotlin 编译器将生成全部所需的 getter 和 setter，如下所示：

```
//Kotlin class declaration
class Fish(var size: Int, var hungry: Boolean)

//class usage in Java
Fish fish = new Fish(12, true);
fish.setSize(7);
System.out.println(fish.getSize());
fish.setHungry(false);
System.out.println(fish.getHungry());
```

实际上，用于访问类属性的语法取决于类使用的实际语言，而不是类声明中的语言，对于在 Android 框架中定义的类，这将支持大量的惯用操作。表 4.2 显示了相关示例。

表 4.2

Java 方法访问语法	Kotlin 属性访问语法
activity.getFragmentManager()	activity.fragmentManager
view.setVisibility(Visibility.GONE)	view.visibility = Visibility.GONE
context.getResources().getDisplayMetrics().density	context.resources.displayMetrics.density

属性访问语法可生成更为简洁的代码，从而降低原 Java 语言的复杂度。需要注意的是，虽然属性访问语法可视作一种较好的替代方案，但 Kotlin 中依然会使用到方法访问语法。

在 Android 框架中，一些方法名称前还会使用到 is 前缀；此时，Boolean 属性也可包含 is 前缀，如下所示：

```
class MainActivity : AppCompatActivity() {

    override fun onDestroy() { // 1
        super.onDestroy()

        isFinishing() // method access syntax
        isFinishing // property access syntax
        finishing // error
    }
}
```

Kotlin 通过 override 修饰符标记重载成员，而非 Java 中的@Override 标记。

虽然使用 finishing 是一种自然、一致的方案，但默认时应避免使用以防止潜在的冲突。

另一个不可使用属性访问语法的场合是：属性仅定义了 setter 且未定义 getter。Kotlin 不支持只写属性，如下所示：

```
fragment.setHasOptionsMenu(true)
fragment.hasOptionsMenu = true // Error!
```

4.2.3　自定义 getter/setter

有些时候，需要对属性应用赋予更大的控制权。当使用属性时，可能需要执行其他辅助操作。例如，在赋予某个字段前对值进行验证、操作的日志记录行为，或者使实例处于无效状态。对此，可制定自定义 setter 和/或 getter。下面向 Fruit 类中加入 ecoRating 属性。大多数时候，需要通过如下方式向类声明中添加该属性：

```
class Fruit(var weight: Double,
            val fresh: Boolean,
            val ecoRating: Int)
```

当确定自定义 getter 和 setter 时，需要在类体中定义一个属性，而非类声明中。具体而言，可将 ecoRating 属性移至类体中，如下所示：

```
class Fruit(var weight: Double, val fresh: Boolean, ecoRating: Int)
{
    var ecoRating: Int = ecoRating
}
```

当在类体中定义属性时，须利用相关值对其进行初始化（甚至空类型数据也需要通过 null 值进行初始化）。对此，可提供默认值，而非使用构造函数参数填充某个属性，如下所示：

```
class Fruit(var weight: Double, val fresh: Boolean) {
    var ecoRating: Int = 3
}
```

另外，还可根据其他属性计算默认值，如下所示：

```
class Apple(var weight: Double, val fresh: Boolean) {
    var ecoRating: Int = when(weight) {
        in 0.5..2.0 -> 5
        in 0.4..0.5 -> 4
        in 0.3..0.4 -> 3
        in 0.2..0.3 -> 2
        else -> 1
    }
}
```

针对不同的 weight 构造函数参数，将设置不同的数值。

当属性在类体中定义时，可忽略类型声明——对应类型可从当前上下文中进行推断，如下所示：

```
class Fruit(var weight: Double) {
    var ecoRating = 3
}
```

下面利用默认行为（等价于之前的属性）设置自定义 getter 和 setter，如下所示：

```
class Fruit(var weight: Double) {
    var ecoRating: Int = 3
    get() {
        println("getter value retrieved")
        return field
    }
    set(value) {
        field = if (value < 0) 0 else value
        println("setter new value assigned $field")
    }
}

// Usage
val fruit = Fruit(12.0)
val ecoRating = fruit.ecoRating
// Prints: getter value retrieved
fruit.ecoRating = 3;
// Prints: setter new value assigned 3
fruit.ecoRating = -5;
// Prints: setter new value assigned 0
```

在 get 和 set 代码块中，可以访问一个特定的变量 field，该变量对应于属性的幕后字段（backing field）。注意，Kotlin 属性声明位置靠近自定义的 getter/setter。这与 Java 明显

不同，并解决了如下问题：字段声明一般位于类文件的顶部，且对应的 getter/setter 处于文件的底部。

因而无法在单一屏幕内对其进行查看，进而导致代码难以阅读。除了位置问题之外，Kotlin 属性与 Java 十分类似。每次从 ecoRating 属性中获取值时，均需要执行 get 代码块；而每次将新值赋予 ecoRating 属性时，则需要执行 set 代码块。

作为只读属性（var），须包含 getter/setter。此时，应显式地定义其中的一个方法，并将另一个用于默认实现。

当每次获取属性值并进行计算时，需要显式地定义 getter，如下所示：

```
class Fruit(var weight: Double) {
    val heavy // 1
    get() = weight > 20
}

// usage
var fruit = Fruit(7.0)
println(fruit.heavy) //prints: false
fruit.weight = 30.5
println(fruit.heavy) //prints: true
```

自 Kotlin 1.1 起，类型将被省略（采用推断方式）。

前述示例使用了 getter，因而每次取值时将计算属性值。通过忽略 getter 这一方式，可针对属性生成默认值。在类构建过程中，该值仅计算一次，且不会产生变化（修改 weight 属性对 isHeavy 属性值不会产生任何影响），如下所示：

```
class Fruit(var weight: Double) {
    val isHeavy = weight > 20
}
var fruit = Fruit(7.0)
println(fruit.isHeavy) // Prints: false
fruit.weight = 30.5
println(fruit.isHeavy) // Prints: false
```

该属性类型包含了幕后字段，其值通常是在对象生成过程中被计算。另外，还可创建不包含幕后字段的读-写属性，如下所示：

```
class Car {
    var usable: Boolean = true
    var inGoodState: Boolean = true

    var crashed: Boolean
    get() = !usable && !inGoodState
    set(value) {
```

```
        usable = false
        inGoodState = false
    }
}
```

此处，属性类型不包含幕后字段，其值通常使用另一个属性计算。

4.2.4　延迟初始化属性

某些时候，已知属性不应为 null，但并不会在声明阶段利用相应值进行初始化。下面考察一个 Android 常见示例，即获取指向布局元素的引用，如下所示：

```
class MainActivity : AppCompatActivity() {

  private var button: Button? = null

  override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      button = findViewById(R.id.button) as Button
  }
}
```

其中，button 在声明阶段无法初始化，因为 MainActivity 布局尚未被初始化。对此，可获取指向 button 的引用（定义于布局内，并位于 onCreate 方法中），但需要将变量声明为可空类型（Button?）。

此类方案缺乏实际操作意义——在 onCreate 方法被调用后，button 实例将一直处于有效状态。然而，客户端仍需要使用安全调用操作符，或者其他空检查操作以对其进行访问。

为了在访问属性时避免执行空检查，需要一种方式告知 Kotlin 编译器，对应变量已在使用前被赋值，但其初始化操作将被延迟。对此，可使用 lateinit 修饰符，如下所示：

```
class MainActivity : AppCompatActivity() {

    private lateinit var button: Button

    override fun onCreate(savedInstanceState: Bundle?) {
        button = findViewById(R.id.button) as Button
        button.text = "Click Me"
    }
}
```

利用标记为 lateinit 的属性时，无须执行空检查即可访问应用程序实例。

lateinit 修饰符告知编译器，对应属性为非空类型，但其初始化操作将被延迟。自然地，若尝试在初始化之前访问属性，应用程序将会抛出 UninitializedPropertyAccess Exception。

变量在声明阶段无法初始化是一类较为常见的情况，通常与视图无关。属性可通过 Dependency Injection 进行初始化，或者通过单元测试的 setup 方法。在此类方案中，无法在构造函数中提供非空值，但仍须避免空检查操作。

延迟属性和框架



当属性通过 Dependency Injection 框架置入时，lateinit 属性十分有用。作为一种较为流行的 Android Dependency Injection 框架，Dagger 使用了@Inject 注解对需要置入的属性予以标记，如下所示：

```
@Inject lateinit var locationManager: LocationManager
```

如前所述，属性不可为 null（将被置入），但 Kotlin 编译器并不理解这一注解。

类似的情形也会出现于另一个常见框架中，即 Mockito，如下所示：

```
@Mock lateinit var mockEventBus: EventBus
```

该变量将被模拟，但会在测试类创建之后的某个时刻发生。

4.2.5　注解属性

Kotlin 从单一属性中可生成多个 JVM 字节码元素（private 字段、getter 以及 setter）。某些时候，框架注解处理器或反射库需要使用特定元素，并定义为公有字段，例如 JUnit 测试框架。当定义 ActivityTestRule，或 Mockito（单元测试的模拟框架）的 Rule 注解时，还会再次面临这一问题，如下所示：

```
@Rule
val activityRule = ActivityTestRule(MainActivity::class.Java)
```

上述代码对 JUnit 未予识别的 Kotlin 属性进行注解，因此，ActivityTestRule 无法实现正常的初始化操作。JUnit 注解处理器期望获得字段或 getter 上的 Rule 注解。这里，存在多种方式可处理此类问题。例如，可作为 Java 字段展示 Kotlin 属性，即利用@JvmField 对其进行注解，如下所示：

```
@JvmField @Rule
val activityRule = ActivityTestRule(MainActivity::class.Java)
```

该字段与底层属性具有相同的可见性。关于@JvmField 注解的使用，存在某些限制条件，如果属性包含幕后字段且为非私有属性，不包含 open、override 或 const 修饰符，也不是一个委托属性，则可利用@JvmField 对其加以注解。

另外，还可直接向 getter 添加注解，如下所示：

```
val activityRule
@Rule get() = ActivityTestRule(MainActivity::class.java)
```

如果不需要定义 getter，仍可向 getter 添加注解。据此，可简单地确定 Kotlin 编译器生成的哪一类元素可被注解，如下所示：

```
@get:Rule
val activityRule = ActivityTestRule(MainActivity::class.Java)
```

4.2.6　内联属性

通过内联修饰符，可优化属性调用。在编译期间，各个属性将被优化。此处并未真正调用某个属性，调用将被属性体所替换，如下所示：

```
inline val now: Long
    get() {
        println("Time retrieved")
        return System.currentTimeMillis()
}
```

关于内联属性，可采用 inline 修饰符。随后，上述代码将编译为：

```
println("Time retrieved")
System.currentTimeMillis()
```

由于无须创建附加对象，因而内联机制可对性能予以改善。另外，方法体将被属性应用替代，因此不会调用 getter。但是，内联机制也包含了自身的局限性——只能应用于不包含幕后字段的属性上。

4.3　构 造 函 数

Kotlin 可定义不包含构造函数的类。除此之外，还可定义一个主构造函数，以及一个或多个次级构造函数，如下所示：

```
class Fruit(val weight: Int) {
    constructor(weight: Int, fresh: Boolean) : this(weight) { }
}

// class instantiation
val fruit1 = Fruit(10)
val fruit2 = Fruit(10, true)
```

次级构造函数不支持属性声明。如果需要使用到一个属性，并由次级构造函数初始化，需要在类体中对其加以声明，并在次级构造函数体中对其进行初始化。下列代码定义了 fresh：

```
class Test(val weight: Int) {
    var fresh: Boolean? = null
    // define fresh property in class body

    constructor(weight: Int, fresh: Boolean) : this(weight) {
        this.fresh = fresh
        // assign constructor parameter to fresh property
    }
}
```

需要注意的是，fresh 属性定义为可空类型——当采用主构造函数生成对象实例时，fresh 属性为 null，如下所示：

```
val fruit = Fruit(10)
println(fruit.weight) // prints: 10
println(fruit.fresh) // prints: null
```

另外，还可将默认值赋予 fresh 属性中，并使其非空，如下所示：

```
class Fruit(val weight: Int) {
    var fresh: Boolean = true

    constructor(weight: Int, fresh: Boolean) : this(weight) {
        this.fresh = fresh
    }
}

val fruit = Fruit(10)
println(fruit.weight) // prints: 10
println(fruit.fresh) // prints: true
```

在定义主构造函数时，各个次级构造函数须隐式或显式地调用主构造函数。其中，隐式调用意味着，直接调用主构造函数；显式调用则表明，调用另一个次级构造函数，且该构造函数调用主构造函数。当调用另一个构造函数时，可使用 this 关键字，如下所示：

```
class Fruit(val weight: Int) {

    constructor(weight: Int, fresh: Boolean) : this(weight) // 1

    constructor(weight: Int, fresh: Boolean, color: String) :
```

```
                this(weight, fresh) // 2
}
```

● 在注释 1 中，调用主构造函数。

● 在注释 2 中，调用次级构造函数。

如果类未包含主构造函数，且超类包含了一个非空的构造函数，那么，次级构造函数须通过 super 关键字初始化基类，或者调用执行此项工作的另一个构造函数，如下所示：

```
class ProductView : View {
   constructor(ctx: Context) : super(ctx)
   constructor(ctx: Context, attrs : AttributeSet) :
               super(ctx, attrs)
}
```

通过@JvmOverloads 注解，上述示例得到了极大的简化。

默认状态下，所生成的构造函数应为公有类型。若不希望生成此类隐式 public 构造函数，可使用 private 或 protected 可见性修饰符，并声明一个空的主构造函数，如下所示：

```
class Fruit private constructor()
```

当改变构造函数的可见性时，须在类定义中显式地使用 constructor 构造函数。另外，当打算注解某个构造函数时，也需要使用到 constructor 关键字。一个常见示例是采用 Dagger（Dependency Injection 框架）的@Inject 注解类构造函数，如下所示：

```
class Fruit @Inject constructor()
```

另外，可见性修饰符和注解还可同时使用，如下所示：

```
class Fruit @Inject private constructor {
    var weight: Int? = null
}
```

4.3.1　属性和构造函数参数

需要注意的是，如果从构造函数属性声明中移除 var/val 关键字，则需要以构造函数参数声明结束。这也表明，属性将被修改为构造函数参数，因此不会生成访问器，且无法在类实例中访问属性，如下所示：

```
class Fruit(var weight:Double, fresh:Boolean)

val fruit = Fruit(12.0, true)
println(fruit.weight)
println(fruit.fresh) // error
```

在上述示例中，由于 fresh 缺少 val 或 var 关键字，因而将会产生错误——此处为构造函数参数，而非类属性（例如 weight）。表 4.3 总结了编译器访问器的生成状况。

表 4.3

类声明	是否生成 getter	是否生成 setter	类型
class Fruit（name:String）	否	否	构造函数参数
class Fruit（val name:String）	是	否	属性
class Fruit（var name:String）	是	是	属性

读者可能会感到疑惑：何时应使用属性，而何时又应使用方法？作为一个较好的准则，建议在下列情况中使用属性（而非方法）：

- 未抛出异常。
- 较小的计算代价（或者首次运行时实现了缓存）。
- 多次调用返回相同的结果。

4.3.2　包含默认参数的构造函数

Java 在早期时候针对对象生成问题存在一系列的缺陷。当对象需要多个参数，且部分参数可选时，通常难以创建一个对象。针对这一问题，存在多种方式可对其加以解决。例如，Telescoping 构造器模式、JavaBeans 模式以及 Builder 模式，各种模式均包含自身的优缺点。

模式用于解决对象生成问题，具体解释如下。

- Telescoping 构造器模式：包含构造器列表的类。其中，每个构造器添加一个新参数。当前，Telescoping 构造器模式视为一种反模式，但 Android 框架仍在多处使用到该模式，例如 android.view.View 类，如下所示：

```
val view1 = View(context)
val view1 = View(context, attributeSet)
val view1 = View(context, attributeSet, defStyleAttr)
```

- JavaBeans 模式：无参构造器以及一个或多个 setter 方法配置对象。该模式的主要问题是，无法知晓全部所需对象是否在一个对象上被调用，或许仅可实现部分构造，如下所示：

```
val animal = Animal()
fruit.setWeight(10)
fruit.setSpeed(7.4)
fruit.setColor("Gray")
```

- Builder 模式：使用另一个对象，即构造器，逐一接收初始化参数；随后，当构造方法被调用时，一次性地返回结果构造对象。例如：

```
android.app.Notification.Builder, or
android.app.AlertDialog.Builder:

Retrofit retrofit = new Retrofit.Builder()
                        .baseUrl("https://api.github.com/")
                        .build();
```

长期以来，构造器应用较为广泛，但默认参数和命名参数语法则是一种更为简洁的方案。下面定义某些默认值，如下所示：

```
class Fruit(weight: Int = 0, fresh: Boolean = true, color:
            String = "Green")
```

通过定义默认参数值，可采用多种方式创建对象，且无须传递全部参数，如下所示：

```
val fruit = Fruit(7.4, false)
println(fruit.fresh) // prints: false

val fruit2 = Fruit(7.4)
println(fruit.fresh) // prints: true
```

当生成对象时，使用包含默认参数的参数语法具有更大的灵活性：可采用任何顺序传递所需参数，且无须定义多个方法和构造函数，如下所示：

```
val fruit1 = Fruit (weight = 7.4, fresh = true, color = "Yellow")
val fruit2 = Fruit (color = "Yellow")
```

4.4 继　　承

如前所述，Any 定义为全部 Kotlin 类型的超类型，即等价于 Java 中的 Object 类型。每个 Kotlin 类均显式或隐式地扩展 Any 类。如果不需要确定父类，则 Any 隐式地设置为该类的父类，如下所示：

```
class Plant // Implicitly extends Any
class Plant : Any // Explicitly extends Any
```

类似于 Java，Kotlin 支持单继承，因此一个类只能包含一个父类，但可实现多个接口。

相比于 Java，Kotlin 中的类和方法，默认状态下均为 final，以防止子类对其进行篡改。基类的修改往往会导致子类中的错误行为，其原因在于，基类修改后的代码将不再与其子类设置相匹配。

这也意味着，类无法被扩展；同时，方法也不可覆写，除非显式地利用 open 关键字对其加以声明。这一点与 Java 的 final 关键字完全不同。

下列代码声明了一个基类 Plant 和一个子类 Tree。

```
class Plant
class Tree : Plant() // Error
```

上述代码尚无法编译——默认状态下，Plant 类处于 final 状态。下面对其进行修改，以使该类处于 open 状态，如下所示：

```
open class Plant
class Tree : Plant()
```

注意，在 Kotlin 中使用冒号即可简单地定义继承，且不必添加 Java 中的 extends 或 implements 关键字。

下面向 Plant 类中添加相关方法和属性，并尝试在 Tree 类中对其进行覆写，如下所示：

```
open class Plant {
    var height: Int = 0
    fun grow(height: Int) {}
}

class Tree : Plant() {
    override fun grow(height: Int) { // Error
        this.height += height
    }
}
```

上述代码同样无法编译。默认条件下，全部方法均处于封闭状态，因而所需覆写的各个方法须显式地标记为 open。下列代码将 grow 方法标记为 open，进而对原代码进行修复。

```
open class Plant {
    var height: Int = 0
    open fun grow(height: Int) {}
}

class Tree : Plant() {
    override fun grow(height: Int) {
```

```
        this.height += height
    }
}
```

采用类似的方式，还可将 height 属性设置为 open 状态，并对其进行覆写，如下所示：

```
open class Plant {
    open var height: Int = 0
    open fun grow(height: Int) {}
}
class Tree : Plant() {
    override var height: Int = super.height
        get() = super.height
        set(value) { field = value}
    override fun grow(height: Int) {
        this.height += height
    }
}
```



当快速对成员进行覆写时，可选取成员所属的类，添加 open 修饰符，并访问需要覆写成员的对应类，运行 override 成员（在 Windows 环境下，快捷键为 Ctrl+O；在 macOS 环境下，快捷键为 Command+O），并选择需要覆写的全部成员。通过这一方式，Android Studio 可生成全部所需代码。

假设全部树木均以相同的方式生长（即针对所有树木，使用相同的生长算法）。对此，应可创建 Tree 类的子类，以对其包含更多控制，同时还可保留生长算法——不允许 Tree 的任何子类覆写该行为。对此，需要显式地将 Tree 类中的 grow 方法标记为 final，如下所示：

```
open class Plant {
    var height: Int = 0

    open fun grow(height: Int) {}
}

class Tree : Plant() {
    final override fun grow(height: Int) {
        this.height += height
    }
}

class Oak : Tree() {
    // 1
}
```

鉴于 grow 方法处于 final 状态，当前尚无法对其进行编译。

关于 open 和 final 的行为，当设置子类中的覆写方法时，须在超类中将其显式地标记为 open。为了确保覆写方法不会被子类再次覆写，须将其标记为 final。

在当前示例中，Plant 类中的 grow 方法未提供任何功能项（仅为空类体），因而无须实例化 Plant 类，且仅视作一个基类。另外，可实例化扩展了 Plant 类的 Tree 类。因此，应将 Plant 类标记为 abstract，以防止其实例化，如下所示：

```
abstract class Plant {
    var height: Int = 0

    abstract fun grow(height: Int)
}

class Tree : Plant() {
    override fun grow(height: Int) {
        this.height += height
    }
}

val plant = Plant()
// error: abstract class can't be instantiated
val tree = Tree()
```

默认条件下，abstract 标记使得方法类处于 open 状态，因而无须将各个成员显式地标记为 open。需要注意的是，当 grow 方法定义为 abstract 时，须移除方法体，abstract 方法不会包含任何方法体。

Android 平台中的某些类使用 Telescoping 构造器，这可视为一种反模式，例如 android.view.View 类。此类情况发生于使用单一构造函数时。但对 android.view.View 子类进行子类化时，可安全地重载 3 个构造函数，其原因在于，该类可在全部场合下正确地工作。通常，自定义 View 类如下所示：

```
class CustomView : View {

    constructor(context: Context?) : this(context, null)

    constructor(context: Context?, attrs: AttributeSet?) :

                this(context, attrs, 0)

    constructor(context: Context?, attrs: AttributeSet?, defStyleAttr:
Int) : super(context, attrs, defStyleAttr) {
```

```
            //...
    }
}
```

代码中针对构造函数引入了大量的样板代码，并托管其他构造函数的调用。针对此类问题，Kotlin 的方案是使用@JvmOverload 注解，如下所示：

```
class KotlinView @JvmOverloads constructor(
    context: Context,
    attrs: AttributeSet? = null,
    defStyleAttr: Int = 0
) : View(context, attrs, defStyleAttr)
```

利用@JvmOverload 注解构造函数将通知编译器，针对包含默认值的各个参数，将在 JVM 字节码中生成额外的重载构造函数。此时，将生成全部所需的构造函数，如下所示：

```
public SampleView(Context context) {
    super(context);
}

public SampleView(Context context, @Nullable AttributeSet attrs) {
    super(context, attrs);
}

public SampleView(Context context, @Nullable AttributeSet attrs, int
defStyleAttr) {
    super(context, attrs, defStyleAttr);
}
```

4.5 接　　口

Kotlin 接口类似于 Java 8 中的接口，并与之前的 Java 版本形成了鲜明的对比。接口通过 interface 关键字定义。下列代码定义了 EmailProvider 接口。

```
interface EmailProvider {
    fun validateEmail()
}
```

当在 Kotlin 中实现该接口时，可使用与扩展类相同的语法，即冒号，且不存在 Java 中的 implements 关键字，如下所示：

```
class User:EmailProvider {
    override fun validateEmail() {
```

```
        // email validation
    }
}
```

这里的问题是，如何同时扩展类并实现一个接口？对此，可简单地将类名置于冒号后，并使用逗号添加一个或多个接口。此处，无须在起始位置放置超类，虽然这可视为一种良好的习惯。

```
open class Person {

    interface EmailProvider {
        fun validateEmail()
    }
    class User: Person(), EmailProvider {
    override fun validateEmail(){
        // email validation
    }
}
```

类似于 Java，Kotlin 类仅可扩展一个类，但却可实现一个或多个接口。除此之外，还可在接口中声明属性，如下所示：

```
interface EmailProvider {
    val email: String
    fun validateEmail()
}
```

全部方法和属性须在实现了接口的类中重载，如下所示：

```
class User() : EmailProvider {

    override val email: String = "UserEmailProvider"

    override fun validateEmail() {
        // email validation
    }
}
```

另外，定义于主构造函数中的属性可用于重载接口中的参数，如下所示：

```
class User(override val email: String) : EmailProvider {
    override fun validateEmail() {
        // email validation
    }
}
```

在接口中，未包含默认实现的全部方法和属性，默认条件下均视为抽象，因而无须显式地将其定义为 abstract。全部抽象方法和属性须通过实现了接口的实体类（非抽象类）予以实现。

然而，还存在另一种方法可在接口中定义方法和属性。Kotlin 与 Java 8 十分相似，并对接口予以重大改进。接口不仅可以定义行为，还可予以实现。这意味着，属性实现的默认方法可通过接口提供。此处唯一的限制条件是，接口无法引用任何幕后字段，即存储某个状态（因为没有合适的地方对其加以存储）。这也是接口和抽象类之间的不同之处。接口不包含任何状态（不具备某个状态）；而抽象类则包含一个状态。如下所示：

```
interface EmailProvider {

    fun validateEmail(): Boolean

    val email: String

    val nickname: String
    get() = email.substringBefore("@")
}
class User(override val email: String) : EmailProvider {
    override fun validateEmail() {
        // email validation
    }
}
```

对于 nickname 属性，EmailProvider 接口提供了默认实现，因而无须在 User 类中对其加以定义，且仍可将当前属性用作定义于类中的其他属性，如下所示：

```
val user = User (" johnny.bravo@test.com")
print(user.nickname) // prints: johnny
```

同样的规则也适用于方法。可简单地在接口中定义包含方法体的某个方法。因此，User 类从接口中获取全部默认实现，且仅须覆写 email 成员——接口中唯一不包含默认实现的成员。如下所示：

```
interface EmailProvider {

    val email: String

    val nickname: String
    get() = email.substringBefore("@")

```

```
    fun validateEmail() = nickname.isNotEmpty()
}

class User(override val email: String) : EmailProvider

// usage
val user = User("joey@test.com")
print(user.validateEmail()) // Prints: true
print(user.nickname) // Prints: joey
```

关于默认实现，有一点须引起注意。一个类无法继承自多个类，但可实现多个接口。例如，可定义两个接口，并包含相同签名和默认实现的方法，如下所示：

```
interface A {
    fun foo() {
        println("A")
    }
}

interface B {
    fun foo() {
        println("B")
    }
}
```

在上述类中，通过覆写（实现了接口的）类中的 foo 方法，即可显式地解决冲突问题，如下所示：

```
class Item : A, B {
    override fun foo() {
        println("Item")
    }
}

// usage
val item = Item()
item.foo() // prints: Item
```

使用尖角括号并确定父接口类型名称，仍然可以通过限定的 super 来调用两个默认的接口实现，如下所示：

```
class Item : A, B {
    override fun foo() {
        val a = super<A>.foo()
```

```
        val b = super<B>.foo()
        print("Item $a $b")
    }
}

// usage
val item = Item()
item.foo()

// Prints: A
           B
           ItemsAB
```

4.6 数 据 类

一种较为常见的情况是，需要构建一个类，旨在存储数据。例如，从服务器或本地数据库获取数据。对应类可表示为应用程序数据模型的构造块，如下所示：

```
class Product(var name: String?, var price: Double?) {

   override fun hashCode(): Int {
       var result = if (name != null) name!!.hashCode() else 0
       result = 31 * result + if (price != null) price!!.hashCode()
       else 0
       return result
   }

   override fun equals(other: Any?): Boolean = when {
       this === other -> true
       other == null || other !is Product -> false
       if (name != null) name != other.name else other.name !=
                         null -> false
       price != null -> price == other.price
       else -> other.price == null
   }

   override fun toString(): String {
       return "Product(name=$name, price=$price)"
   }
}
```

在 Java 中，需要生成大量的冗余 getter/setter、hashCode 以及 equals 方法。对此，Android

Studio 可为我们创建大多数代码，但代码维护仍是一个重要的问题。在 Kotlin 中，可定义称之为数据类的特定类，即向类声明中添加 data 关键字，如下所示：

```
class Product(var name: String, var price: Double)
// normal class

data class Product(var name: String, var price: Double)
// data class
```

数据类通过 Kotlin 编译器生成的方法，向类中添加了附加功能，相关方法包括 equals、hashCode、toString、copy 以及多个 componentN 方法。此处的限制条件则是数据类无法标记为 abstract、inner 和 sealed。下面详细讨论添加了数据修饰符的各种方法。

4.6.1　equals 和 hashCode 方法

当处理数据类时，通常需要比较两个实例架构间的等同性（包含相同的数据。但不必是同一个实例）。相应地，需要检测 User 类实例是否等于另一个 User 类实例；或者两个产品实例（product instance）。常见的对象相等检测模式是使用 equals 方法，其内部采用了 hashCode 方法，如下所示：

```
product.equals(product2)
```

针对 hashCode 的重载实现，总体原则是：两个相等的对象（根据 equals 实现）应包含相同的哈希码，背后的原因可解释为：考虑到性能问题，hashCode 先于 equals 进行比较——与对象中的各个字段相比，比较哈希码则具有较好的性能。

若 hashCode 相等，则 equals 方法负责检测两个对象是否为同一实例、同一类型，随后通过比较全部有效字段验证相等性。若出现一个字段不等，则两个对象即视为不相等。另一种方法可解释为：若两个对象具有相同的 hashCode，且全部（比较后的）有效字段包含相同值，则二者视为相等。下面考察包含两个字段（name 和 price）的 Java 类示例，如下所示：

```
public class Product {

    private String name;
    private Double price;

    public Product(String name, Double price) {
        this.name = name;
        this.price = price;
    }

    @Override
```

```
    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;
     result = 31 * result + (price != null ?
                                price.hashCode() : 0);
        return result;
}

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }

        Product product = (Product) o;

        if (name != null ? !name.equals(product.name) :
        product.name != null) {
            return false;
        }
        return price != null ? price.equals(product.price) :
        product.price == null;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Double getPrice() {
        return price;
    }

    public void setPrice(Double price) {
        this.price = price;
    }
}
```

该方案广泛应用于 Java 和其他 OOP 程序设计语言中。早期，程序员须针对需要比较的每个类通过手动方式编写代码并予以维护，以确保其正确性并针对各个有效值进行比较。

当前，现代 IDE 均可生成此类代码，并对相应的方法进行更新，例如 Android Studio。读者无须手工编写此类代码，当仍须对此予以适当维护——确保全部所需字段均经由 equals 方法进行比较。某些时候，我们并不了解是否为 IDE 生成的标准代码，或者是调整后的版本——对于各个 Kotlin 数据类，相关方法均由编译器自动生成，因而并不会存在此类问题。下列代码在 Kotlin 中定义了 Product 类，其中包含了之前 Java 类中的全部方法。

```
data class Product(var name: String, var price: Double)
```

该类包含了定义于 Java 中的全部方法，但存在大量的样板代码需要维护。

第 2 章曾讨论到，使用结构相等操作符将调用底层 equals，这也表明，此处可方便、安全地比较 Product 类实例，如下所示：

```
data class Product(var name:String, var price:Double)

val productA = Product("Spoon", 30.2)
val productB = Product("Spoon", 30.2)
val productC = Product("Fork", 17.4)

print(productA == productA) // prints: true
print(productA == productB) // prints: true
print(productB == productA) // prints: true
print(productA == productC) // prints: false
print(productB == productC) // prints: false
```

默认状态下，hashCode 和 equals 方法根据主构造函数中声明的各项属性而生成。在大多数场合下，这两种方法已然足够；但若需要实施更多控制，则应在数据类中重载这一类方法。此时，默认实现将不再由编译器生成。

4.6.2　toString 方法

所生成的方法包含主构造函数中声明的名字和全部属性值，如下所示：

```
data class Product(var name:String, var price:Double)
val productA = Product("Spoon", 30.2)
println(productA) // prints: Product(name=Spoon, price=30.2)
```

实际上，可将数据输出至控制台或日志文件中，而不是类似于 Java 中的类名和内存地址（Person@a4d2e77）。由于具有人类可读的适宜格式，因而可简化调试处理过程。

4.6.3　copy 方法

默认条件下，Kotlin 编译器可生成 copy 方法，进而可方便地创建某个对象的副本，如下所示：

```
data class Product(var name: String, var price: Double)
val productA = Product("Spoon", 30.2)
print(productA) // prints: Product(name=Spoon, price=30.2)
val productB = productA.copy()
print(productB) // prints: Product(name=Spoon, price=30.2)
```

Java 并不包含命名参数语法，因此，当调用 copy 方法的 Java 代码时，需要传递全部参数（参数顺序对应于主构造函数中定义的属性顺序）。在 Kotlin 中，该方案可减少对 copy 构造函数或 copy 工厂的需求。

- copy 构造函数接收单一参数，对应类型为包含当前构造函数的类，并返回该类的新实例，如下所示：

```
val productB = Product(productA)
```

- copy 过程表示为一类静态工厂，并接收单一参数，其类型为包含该工厂的类，并返回该类的新实例，如下所示：

```
val productB = ProductFactory.newInstance(productA)
```

copy 方法接收与主构造函数中声明的属性对应的参数。当与默认参数语法结合使用时，可提供全部或部分属性，进而生成调整后的实例副本，如下所示：

```
data class Product(var name:String, var price:Double)

val productA = Product("Spoon", 30.2)
print(productA) // prints: Product(name=Spoon, price=30.2)

val productB = productA.copy(price = 24.0)
print(productB) // prints: Product(name=Spoon, price=24.0)

val productC = productA.copy(price = 24.0, name = "Knife")
print(productB) // prints: Product(name=Knife, price=24.0)
```

这可视作一种灵活的对象副本创建方式，进而可明晰副本与原始实例间的不同之处。另外一方面，程序设计方案强调了不可变性这一概念，并可利用 copy 方法的无参数调用方便地予以实现，如下所示：

```
// Mutable object - modify object state
data class Product(var name:String, var price:Double)

var productA = Product("Spoon", 30.2)
productA.name = "Knife"

// immutable object - create new object instance
data class Product(val name:String, val price:Double)

var productA = Product("Spoon", 30.2)
productA = productA.copy(name = "Knife")
```

此处并未定义可变属性（var）并调整对象状态；相反，可定义不可变属性（val），使对象处于不可变状态，通过获取包含变化值的副本对其进行操作。该方案降低了多线程应用程序中对数据同步的需求，以及与其相关的潜在错误数量，因为不可变对象可在线程之间自由共享。

4.6.4 解构声明

某些时候，可将对象重构为多个变量，该语法称作解构声明，如下所示：

```
data class Person(val firstName: String, val lastName: String,
                  val height: Int)
val person = Person("Igor", "Wojda", 180)
var (firstName, lastName, height) = person
println(firstName) // prints: "Igor"
println(lastName) // prints: "Wojda"
println(height) // prints: 180
```

解构声明可一次性地创建多个变量。在上述代码中，将生成 firstName、lastName 和 height 变量。从底层来看，编译器将生成如下代码：

```
val person = Person("Igor", "Wojda", 180)
var firstName = person.component1()
var lastName = person.component2()
var height = person.component3()
```

对于数据类主构造函数中声明的各个属性，Kotlin 编译器生成单一的 componentN 方法。component 方法后缀对应于主构造函数中声明的属性顺序。因此，firstName 对应于 component1，lastName 对应于 component2，height 对应于 component3。实际上，可直接在 Person 类上调用此类方法以获取属性值，但这里并无此必要——对应的名称不包含实际含义，代码也难以阅读和维护。针对对象的解构，可将此类方法留与编译器，并使用诸如 person.firstName 这一类属性访问语法。

另外，当使用下划线时，还可忽略一个或多个属性，如下所示：

```
val person = Person("Igor", "Wojda", 180)
var (firstName, _, height) = person
println(firstName) // prints: "Igor"
println(height) // prints: 180
```

代码仅须生成两个变量，即 firstName 和 height。其中，lastName 将被忽略。编译器生成的代码如下所示：

```
val person = Person("Igor", "Wojda", 180)
var firstName= person.component1()
var height = person.component3()
```

同时，还可像 String 那样对简单类型进行解构，如下所示：

```
val file = "MainActivity.kt"
val (name, extension) = file.split(".", limit = 2)
```

最后，解构声明还可与 for 循环结合使用，如下所示：

```
val authors = listOf(
    Person("Igor", "Wojda", 180),
    Person("Marcin", "Moskała", 180)
)

println("Authors:")
for ((name, surname) in authors) {
    println("$name $surname")
}
```

4.7　操作符重载

Kotlin 中包含了预定义操作符集合，并使用特定的符号表达方式（例如+、*等）和优先级。大多数操作符可直接转换为方法调用；某些操作符则可转换为更加复杂的表达式。表 4.4 显示了 Kotlin 中的操作符列表。

表 4.4

操作符标记	对应的方法/表达式
a + b	a.plus(b)
a–b	a.minus(b)
a * b	a.times(b)

续表

操作符标记	对应的方法/表达式
a / b	a.div(b)
a % b	a.rem(b)
a..b	a.rangeTo(b)
a += b	a.plusAssign(b)
a -= b	a.minusAssign(b)
a *= b	a.timesAssign(b)
a /= b	a.divAssign(b)
a %= b	a.remAssign(b)
a++	a.inc()
a--	a.dec()
a in b	b.contains(a)
a !in b	!b.contains(a)
a[i]	a.get(i)
a[i, j]	a.get(i, j)
a[i_1, ..., i_n]	a.get(i_1, ..., i_n)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i_1, ..., i_n] = b	a.set(i_1, ..., i_n, b)
a()	a.invoke()
a(i)	a.invoke(i)
a(i, j)	a.invoke(i, j)
a(i_1, ..., i_n)	a.invoke(i_1, ..., i_n)
a == b	a?.equals(b) ?: (b === null)
a != b	!(a?.equals(b) ?: (b === null))
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a >= b	a.compareTo(b) >= 0
a <= b	a.compareTo(b) <= 0

Kotlin 编译器将表示特定操作符（表中的左列）的标记转换为将被调用的对应方法或表达式（右列）。

针对各项操作符，可提供自定义实现，也就是说，在与 operator 标记相符的类 operator 方法中对其加以使用。下面定义一个包含 x，y 属性，以及 plus 和 times 操作符的 Point 类。

```
data class Point(var x: Double, var y: Double) {
   operator fun plus(point: Point) = Point(x + point.x, y+ point.y)

   operator fun times(other: Int) = Point(x * other, y * other)
}

// usage
var p1 = Point(2.9, 5.0)
var p2 = Point(2.0, 7.5)

println(p1 + p2) // prints: Point(x=4.9, y=12.5)
println(p1 * 3) // prints: Point(x=8.7, y=21.0)
```

通过定义 plus 和 times 操作符，可在任意 Point 实例上执行加法和乘法操作。每次调用+或*操作符时，Kotlin 调用对应的操作符方法 plus 或 times。实际上，编译器将生成下列方法调用：

```
p1.plus(p2)
p1.times(3)
```

在当前示例中，向 plus 操作符传递了另一个 point 实例，该类型并非是强制性的。操作符方法实际上并未重载源自超类的方法，因而未包含基于特定参数和特定类型的声明。同时，无须继承特定的 Kotlin 类型以重载操作符。全部所需内容表示为一个方法，且包含了标记为 operator 的签名。Kotlin 编译器通过运行对应当前操作符的相关方法执行其余内容。实际上，可定义包含相同名称和不同参数类型的多个操作符，如下所示：

```
data class Point(var x: Double, var y: Double) {
   operator fun plus(point: Point) = Point(x + point.x, y +point.y)

   operator fun plus(vector:Double) = Point(x + vector, y + vector)
}

var p1 = Point(2.9, 5.0)
var p2 = Point(2.0, 7.5)

println(p1 + p2) // prints: Point(x=4.9, y=12.5)
println(p1 + 3.1) // prints: Point(x=6.0, y=10.1)
```

两个操作符均工作良好——Kotlin 编译器可选取操作符的适当重载结果。另外，许多基本操作符还包含对应的复合赋值操作符（plus 包含 plusAssign，times 包含 times Assign，等等）。因此，当定义诸如+这一类操作符时，Kotlin 支持+操作和+=操作，如下所示：

```
var p1 = Point(2.9, 7.0)
var p2 = Point(2.0, 7.5)

p1 += p2
println(p1) // prints: Point(x=4.9, y=14.5)
```

在性能要求较为严格的场合下，应注意不同方案之间的差异。复合赋值操作符（例如+=操作符）包含 Unit 返回类型，因而仅调整现有对象的状态；而基本操作符（例如+操作符）通常返回某个对象的新实例，如下所示：

```
var p1 = Wallet(39.0, 14.5)
p1 += p2 // update state of p1
val p3 = p1 + p2 //creates new object p3
```

当定义包含相同参数类型的 plus 和 plusAssign 操作符，并尝试使用 plusAssign（复合）操作符时，编译器将抛出错误，其原因在于编译器并不知晓应调用哪一个方法，如下所示：

```
data class Point(var x: Double, var y: Double) {
    init {
        println("Point created $x.$y")
    }
    operator fun plus(point: Point) = Point(x + point.x, y + point.y)

    operator fun plusAssign(point:Point) {
        x += point.x
        y += point.y
    }
}

// usage
var p1 = Point(2.9, 7.0)
var p2 = Point(2.0, 7.5)
val p3 = p1 + p2
p1 += p2 // Error: Assignment operations ambiguity
```

操作符重载同样适用于定义 Java 中的类。全部工作仅须定义包含适当签名和名称的方法（对应于操作符的方法名）。Kotlin 编译器将操作符应用转换为该方案。另外，Java 中不存在操作符修饰符，因而 Java 类中对其也不予涉及，如下所示：

```
// Java
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
```

```
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
    public Point plus(Point point) {
        return new Point(point.getX() + x, point.getY() + y);
    }
}
// Main.kt
val p1 = Point(1, 2)
val p2 = Point(3, 4)
val p3 = p1 + p2;
println("$x:{p3.x}, y:${p3.y}") //prints: x:4, y:6
```

4.8 对象声明

在 Java 中，存在多种方式可声明单例。一种较为常见的方式是定义一个包含私有构造函数的类，并通过静态工厂方法获取实例，如下所示：

```
public class Singleton {

    private Singleton() {
    }

    private static Singleton instance;

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }

        return instance;
    }
}
```

上述代码针对单例线程工作良好，但并不具备线程安全性。因此，在某些场合下，可创建两个 Singleton 实例。对此，存在多种方式可对其进行修正。例如，下列代码采用了 synchronized 代码块。

```
// synchronized
public class Singleton {

    private static Singleton instance = null;

    private Singleton(){
    }

    private synchronized static void createInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
    }

    public static Singleton getInstance() {
        if (instance == null) createInstance();
        return instance;
    }
}
```

然而，上述代码仍稍显冗余。在 Kotlin 中，针对称作对象声明的单例构建，存在一种特定的语言结构，并可采用更加简单的方式实现相同结果。定义对象的过程与定义类十分类似，唯一的差别在于使用 object 关键字，而非 class 关键字，如下所示：

```
object Singleton
```

另外，还可采用与类相同的方式，向对象声明添加方法和属性，如下所示：

```
object SQLiteSingleton {
    fun getAllUsers(): List<User> {
        //...
    }
}
```

该方法与任意 Java 静态类的方法访问方式均保持相同，如下所示：

```
SQLiteSingleton.getAllUsers()
```

对象声明可实现延迟初始化，并可嵌套于其他对象声明或非内部类中。另外，对象声明不可被赋予某个变量中。

4.9 对象表达式

对象表达式等价于 Java 中的匿名类，用于实例化继承自某个类或实现了某个接口的对象。对此，较为经典的用例是需要定义实现了某个接口的对象。下列代码显示了在 Java 中，如何实现 ServiceConnection 接口，并将其赋予某个变量。

```
ServiceConnection serviceConnection = new ServiceConnection() {
    @Override
    public void onServiceDisconnected(ComponentName name) {
        ...
    }

    @Override
    public void onServiceConnected(ComponentName name,
        IBinder service)
    {
        ...
    }
}
```

针对上述实现，Kotlin 中最为接近的实现方式如下所示：

```
val serviceConnection = object: ServiceConnection {

 override fun onServiceDisconnected(name: ComponentName?) { }

 override fun onServiceConnected(name: ComponentName?,
     service: IBinder?) { }
}
```

当前示例采用了一个对象表达式，并生成实现了 ServiceConnection 接口的匿名类实例。除此之外，对象表达式还可进一步扩展类。下列代码显示了如何生成抽象类 Broadcast Receiver 的实例。

```
val broadcastReceiver = object : BroadcastReceiver() {
   override fun onReceive(context: Context, intent: Intent) {
       println("Got a broadcast ${intent.action}")
   }
}

val intentFilter = IntentFilter("SomeAction");
registerReceiver(broadcastReceiver, intentFilter)
```

对象表达式可生成匿名类型的对象（实现了某个接口）并扩展某个类，此外，还可据此方便地处理与适配器模式相关的某些有趣问题。



适配器设计模式允许将一个类的接口转换为客户期望的接口，从而使不兼容的类能够协同工作。

下列代码定义了一个 Player 接口，以及一个以 Player 作为参数的接口。

```
interface Player {
    fun play()
}

fun playWith(player: Player) {
    print("I play with")
    player.play()
}
```

另外，此处使用了公共库中定义的 VideoPlayer 类，该类包含了一个定义的 play 方法，但并未实现当前的 Player 接口，如下所示：

```
open class VideoPlayer {
    fun play() {
        println("Play video")
    }
}
```

VideoPlayer 满足全部接口需求条件，但无法作为 Player 被传递——未实现当前接口。当用作 player 时，需要定义一个适配器。在当前示例中，须将其实现为匿名类型对象，同时实现了 Player 接口，如下所示：

```
val player = object: VideoPlayer(), Player { }
playWith(player)
```

这里，无须定义 VideoPlayer 子类即可解决当前问题。另外，可在对象表达式中实现自定义方法，如下所示：

```
val data = object {
    var size = 1
    fun update() {
        //...
    }
}
```

```
data.size = 2
data .update()
```

作为一种非常简便的方法，代码自定义了匿名对象，这在 Java 中尚无法体现。当在 Java 中定义类似的类型时，须自定义接口。当前，可向 VideoPlayer 类添加某种操作行为，并完整实现 Player 接口，如下所示：

```
open class VideoPlayer {
    fun play() {
        println("Play video")
    }
}

interface Player{
    fun play()
    fun stop()
}

// usage
val player = object: VideoPlayer(), Player {

    var duration:Double = 0.0

    fun stop() {
        println("Stop video")
    }
}

player.play() // println("Play video")
player.stop() // println("Stop video")
player.duration = 12.5
```

其中，可调用定义于 VideoPlayer 类和表达式对象中的匿名对象（player）方法。

4.10　伴 生 对 象

与 Java 不同，Kotlin 缺乏定义静态方法的能力，但可定义与某个类关联的对象。换而言之，某个对象仅初始化一次，因而仅存在对象的唯一实例，并在特定类的全部实例间共享其状态。当单例对象与同名类关联时，称作伴生对象，对应类称作该类的伴生类，如图 4.1 所示。

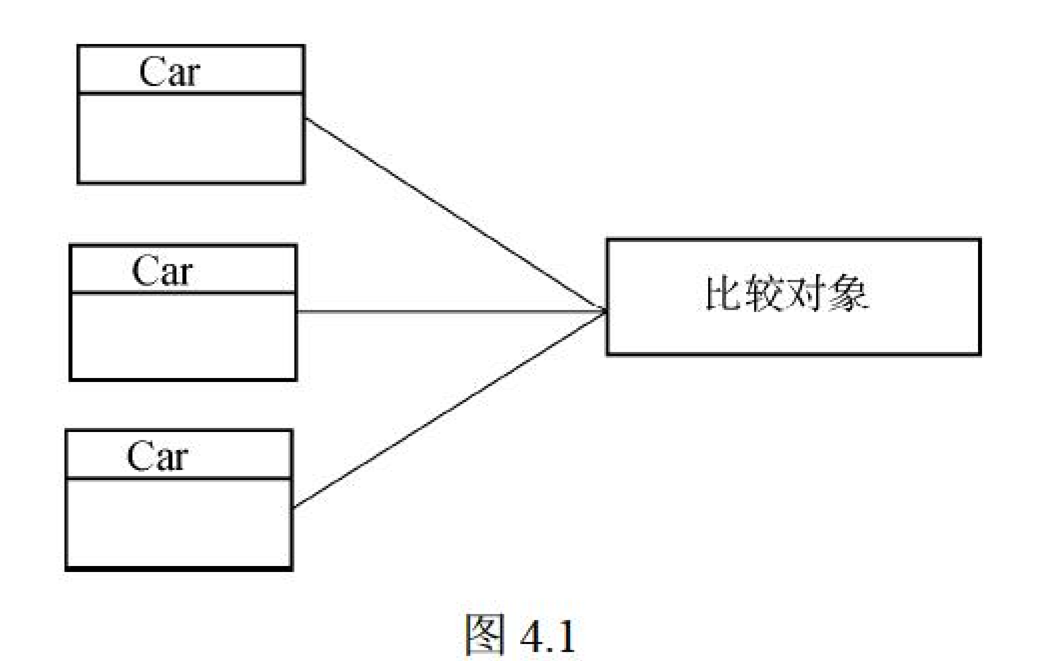


图 4.1

图 4.1 中显示了 Car 类的 3 个实例，并共享某个对象的单一实例。

定义于伴随对象中的成员（例如方法和属性），可采用与 Java 访问静态字段和方法类似的方式予以访问。伴生对象的主要功能是，令代码与类相关联，而不必针对特定的类实例。相对于 Java 中的静态成员，一种较好的成员定义方式是工厂模式，进而生成类实例方法。当定义最简单的伴生对象时，须定义一个独立的代码块，如下所示：

```
class ProductDetailsActivity {

    companion object {
    }
}
```

下面定义一个 start 方法，并以一种较为简单的方式启动某项操作（activity），如下所示：

```
// ProductDetailsActivity.kt
class ProductDetailsActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val product = intent.getParcelableExtra<Product>
            (KEY_PRODUCT) // 3
        //...
    }

    companion object {

        const val KEY_PRODUCT = "product" // 1

        fun start(context: Context, product: Product) { // 2
            val intent = Intent(context,
                ProductDetailsActivity::class.java)
            intent.putExtra(KEY_PRODUCT, product) // 3
```

```
                context.startActivity(intent)
            }
        }
}

// Start activity
ViewProductActivity.start(context, productId) // 2
```

- 对于注释 1，仅存在 key 的单实例。
- 无须生成对象实例即可调用 start 方法，这一点类似于 Java 中的静态方法。
- 实例创建后获取数值。

需要注意的是，可在操作生成之前调用 start。下面利用伴生对象跟踪 Car 类实例的创建方式。对此，需要定义包含私有 setter 的 count 属性；除此之外，还可定义为顶级属性。但较好的方法是将其置于伴生对象中，其原因在于：不希望在当前类外部修改 count，如下所示：

```
class Car {
    init {
        count++;
    }

    companion object {
        var count:Int = 0
        private set
    }
}
```

该类可访问定义于伴生对象中的全部方法和属性，但伴生对象则无法访问类内容。伴生对象被赋予一个类中，而非特定的实例，如下所示：

```
println(Car.count) // Prints 0
Car()
Car()
println(Car.count) // Prints: 2
```

当直接访问伴生对象实例时，可使用类名。



可通过稍显复杂的语法访问伴生对象，例如 Car.Companion.count。但大多数时候则无此必要，除非需要从 Java 代码中访问 companion。

伴生对象表示为伴生类创建的单体，并保持其静态属性。companion 对象采用延迟初始化方式，这也意味着，companion 对象在首次需要时予以实例化。例如，访问其成员时，

或者包含 companion 对象的类实例被创建时。当标识何时创建 Car 类实例及其对应的伴生对象时，须添加两个初始化代码块，分别对应于 Car 类和伴生对象。

伴生对象中初始化代码块的工作方式与类相同——当创建实例时被执行，如下所示：

```
class Car {
    init {
         count++;
         println("Car created")
    }

    companion object {
        var count: Int = 0
        init {
            println("Car companion object created")
        }
    }
}
```

类初始化代码块等价于 Java 的构造方法体；而编译对象初始化代码块则与 Java 静态初始化代码块等价。当前，count 属性可通过任意客户端进行更新——该属性可在 Car 外部访问，稍后将对这一问题进行修正。下面尝试访问 Car 伴生对象的类成员，如下所示：

```
Car.count // Prints: Car companion object created
Car() // Prints: Car created
```

当访问定义于伴生对象的 count 属性时，将引发该对象的创建过程。需要注意的是，此处并未生成 Car 类实例。稍后将讨论，当生成 Car 类时，companion 对象已创建完毕。下面考察 Car 类的实例化操作，并于随后访问 companion 对象，如下所示：

```
Car()
// Prints: Car companion object created
// Prints: Car created

Car() // Prints: Car created
Car.count
```

伴生对象随首个 Car 类实例而被创建，因此，当生成其他用户类实例时，针对该类的 companion 对象已然存在，且无须创建。

注意，前述实例化机制描述了两个单独示例，这两种情况在单一程序中均不可能成立——仅可存在唯一的类 companion 对象实例，并在需要时首次创建。

companion 对象还可定义函数、实现相关接口，甚至是扩展类。对此，可定义一个伴生对象，在其中定义包含附加功能项的静态方法，并针对测试目的予以重载实现，如下所示：

```
 abstract class Provider<T> { // 1
     abstract fun creator(): T // 2

     private var instance: T? = null // 3
     var override: T? = null // 4

     fun get(): T = override ?: instance ?: creator().also { instance = it
} //5
 }
```

- 针对注释 1，Provider 定义为泛型类。
- 针对注释 2，抽象函数用于创建实例。
- 针对注释 3，对应字段用于保存创建后的实例。
- 针对注释 4，该字段用于测试目的，并提供实例的替代实现。
- 针对注释 5，该函数返回重载后的实例（若已设置），或者创建后的实例化操作，抑或通过 create 方法生成实例，并以此填充实例字段。

根据此类实现，即可定义包含默认静态构造器的接口，如下所示：

```
  interface MarvelRepository {

    fun getAllCharacters(searchQuery: String?):
Single<List<MarvelCharacter>>

     companion object : Provider<MarvelRepository>() {
         override fun creator() = MarvelRepositoryImpl()
     }
}
```

当获取实例时，需要使用

```
MarvelRepository.get()
```

出于测试目的，如果需要确定其他实例（例如 Espresso 测试），通常可采用对象表达式予以确定，如下所示：

```
MarvelRepository.override = object : MarvelRepository {
    override fun getAllCharacters(searchQuery: String?):
    Single<List<MarvelCharacter>> {
        //...
    }
}
```

伴生对象在 Kotlin Android 开发中十分常见，常用于定义 Java 中的静态数据元素（常量字段、静态创建方法等），但也会提供其他一些附加功能项。

4.11 枚 举 类

枚举类型（enum）定义为一种数据类型，并由一组命名值构成。当定义 enum 类型时，需要向类声明中添加 enum 关键字，如下所示：

```
enum class Color {
    RED,
    ORANGE,
    BLUE,
    GRAY,
    VIOLET
}

val favouriteColor = Color.BLUE
```

若将字符串解析为 enum，可使用 valueOf 方法（与 Java 类似），如下所示：

```
val selectedColor = Color.valueOf("BLUE")
println(selectedColor == Color.BLUE) // prints: true
```

或者使用 Kotlin 中的帮助方法，如下所示：

```
val selectedColor = enumValueOf<Color>("BLUE")
println(selectedColor == Color.BLUE) // prints: true
```

当显示 Color 枚举值中的全部值时，可使用 values 函数（与 Java 类似），如下所示：

```
for (color in Color.values()) {
    println("name: ${it.name}, ordinal: ${it.ordinal}")
}
```

或者使用 Kotlin 中的 enumerateValues 帮助函数，如下所示：

```
for (color in enumValues<Color>()) {
    println("name: ${it.name}, ordinal: ${it.ordinal}")
}

// Prints:
name: RED, ordinal: 0
name: ORANGE, ordinal: 1
name: BLUE, ordinal: 2
name: GRAY, ordinal: 3
name: VIOLET, ordinal: 4
```

另外，enum 类型还可包含构造方法，以及与各个 enum 常量关联的自定义数据。下面

利用 red、green、blue 颜色分量值添加属性，如下所示：

```
enum class Color(val r: Int, val g: Int, val b: Int) {
    RED(255, 0, 0),
    ORANGE(255, 165, 0),
    BLUE(0, 0, 255),
    GRAY(49, 79, 79),
    VIOLET(238, 130, 238)
}

val color = Color.BLUE
val rValue =color.r
val gValue = color.g
val bValue = color.b
```

据此，可针对各种颜色定义 RGB 值计算函数。

注意，如果在最后一个常量（VIOLET）之后添加分号，将把常量定义从成员定义中分离开来，这在 Kotlin 中较少出现，如下所示：

```
enum class Color(val r: Int, val g: Int, val b: Int) {
    BLUE(0, 0, 255),
    ORANGE(255, 165, 0),
    GRAY(49, 79, 79),
    RED(255, 0, 0),
    VIOLET(238, 130, 238);

    fun rgb() = r shl 16 + g shl 8 + b
}
fun printHex(num: Int) {
    println(num.toString(16))
}

printHex(Color.BLUE.rgb()) // Prints: ff
printHex(Color.ORANGE.rgb()) // Prints: ffa500
printHex(Color.GRAY.rgb()) // Prints: 314f4f
```

rgb()方法针对某个特定枚举值访问 r、g、b，并针对每个 enum 数据元素单独计算对应值。另外，还可通过 init 代码块向枚举构造方法参数中加入验证机制；在 Kotlin 中，则需要使用到下列函数：

```
enum class Color(val r: Int, val g: Int, val b: Int) {
    BLUE(0, 0, 255),
    ORANGE(255, 165, 0),
    GRAY(49, 79, 79),
    RED(255, 0, 0),
```

```
    VIOLET(238, 130, 238);

    init {
        require(r in 0..255)
        require(g in 0..255)
        require(b in 0..255)
    }

    fun rgb() = r shl 16 + g shl 8 + b
}
```

定义错误的枚举值将会导致异常，如下所示：

```
GRAY(33, 33, 333) // IllegalArgumentException: Failed requirement
```

某些时候，需要将不同的操作行为关联至各个常量上。对此，可定义抽象方法或属性，并在各个枚举代码块中对其进行重载。下面定义枚举 Temperature 和 temperature 属性：

```
enum class Temperature { COLD, NEUTRAL, WARM }

enum class Color(val r: Int, val g: Int, val b: Int) {
    RED(255, 0, 0) {
        override val temperature = Temperature.WARM
    },
    ORANGE(255, 165, 0) {
        override val temperature = Temperature.WARM
    },
    BLUE(0, 0, 255) {
        override val temperature = Temperature.COLD
    },
    GRAY(49, 79, 79) {
        override val temperature = Temperature.NEUTRAL
    },
    VIOLET(238, 130, 238 {
        override val temperature = Temperature.COLD
    };

     init {
        require(r in 0..256)
        require(g in 0..256)
        require(b in 0..256)
    }

    fun rgb() = (r * 256 + g) * 256 + b
```

```
    abstract val temperature: Temperature
}

println(Color.BLUE.temperature) //prints: COLD
println(Color.ORANGE.temperature) //prints: WARM
println(Color.GRAY.temperature) //prints: NEUTRAL
```

目前，每种颜色步进包含了 RGB 信息，而且还涵盖了描述其温度的附加枚举值。此处已添加了一项属性，采用类似的方式还可向每个枚举元素添加自定义方法。

4.12　命名方法的中缀调用

中缀调用可视作 Kotlin 的特征之一，进而可生成流畅且具有可读性的代码。中缀调用使得代码编写方式更接近于人类语言。第 2 章曾讨论了中缀方法，并可方便地创建 Pair 类实例，如下所示：

```
var pair = "Everest" to 8848
```

Pair 类表示为包含两个值的泛型对，在该类中，数值并不存在具体含义，因而可用于各种用途。这里，Pair 定义为一个数据类，因而包含了全部数据类方法（equals、hashCode、component1 等）。下列代码显示了 Kotlin 标准库中 Pair 类的定义。

```
public data class Pair<out A, out B>( // 1
   public val first: A,
   public val second: B
) : Serializable {

   public override fun toString(): String = "($first, $second)"
   // 2
}
```

- 针对注释 1，泛型背后所用的 out 修饰符的含义将在第 6 章加以讨论。
- 针对注释 2，Pair 包含了一个自定义 toString 方法，其实现使得输出语法更具可读性；而第一个和第二个名称在大多数使用环境下并不具备实际含义。

在深入讨论中缀方法的定义方式之前，下面首先将所述代码转换为较为熟悉的形式。每种中缀方法均可像其他方法那样使用，如下所示：

```
val mountain = "Everest";
var pair = mountain.to(8848)
```

实际上，中缀可简单地表示为方法调用，且无须使用点号操作符和调用操作符（即括号）。中缀标识仅是外观不同，但仍可视为常规的方法调用。在之前的例子中，可简单地在 String 类接口上调用 to 方法。这里，to 表示为扩展函数，第 7 章将对此予以解释。当前，可假设 to 表示为 String 类中的一个方法，并返回包含自身和传递参数的 Pair 实例。随后，可在返回的 Pair 上像其他数据类对象那样进行操作，如下所示：

```
val mountain = "Everest";
var pair = mountain.to(8848)
println(pair.first) //prints: Everest
println(pair.second) //prints: 8848
```

在 Kotlin 中，该方法仅在包含单一参数时可实现中缀化。另外，中缀标识并不会自动出现——须显式地将该方法标记为 infix。下面利用中缀方法定义 Point 类，如下所示：

```
data class Point(val x: Int, val y: Int) {
    infix fun moveRight(shift: Int) = Point(x + shift, y)
}

Usage example:

val pointA = Point(1,4)
val pointB = pointA moveRight 2
println(pointB) //prints: Point(x=3, y=4)
```

需要注意的是，此处创建了一个新的 Point 实例，并可对现有实例进行修改（若为可变类型）。另外，中缀也常与不可变类型结合使用。

我们还可将 infix 方法和枚举值结合使用，进而实现更为流畅的语法。下面考察经典的纸牌游戏中的每幅纸牌，其中包括梅花、方片、红心和黑桃，如图 4.2 所示。

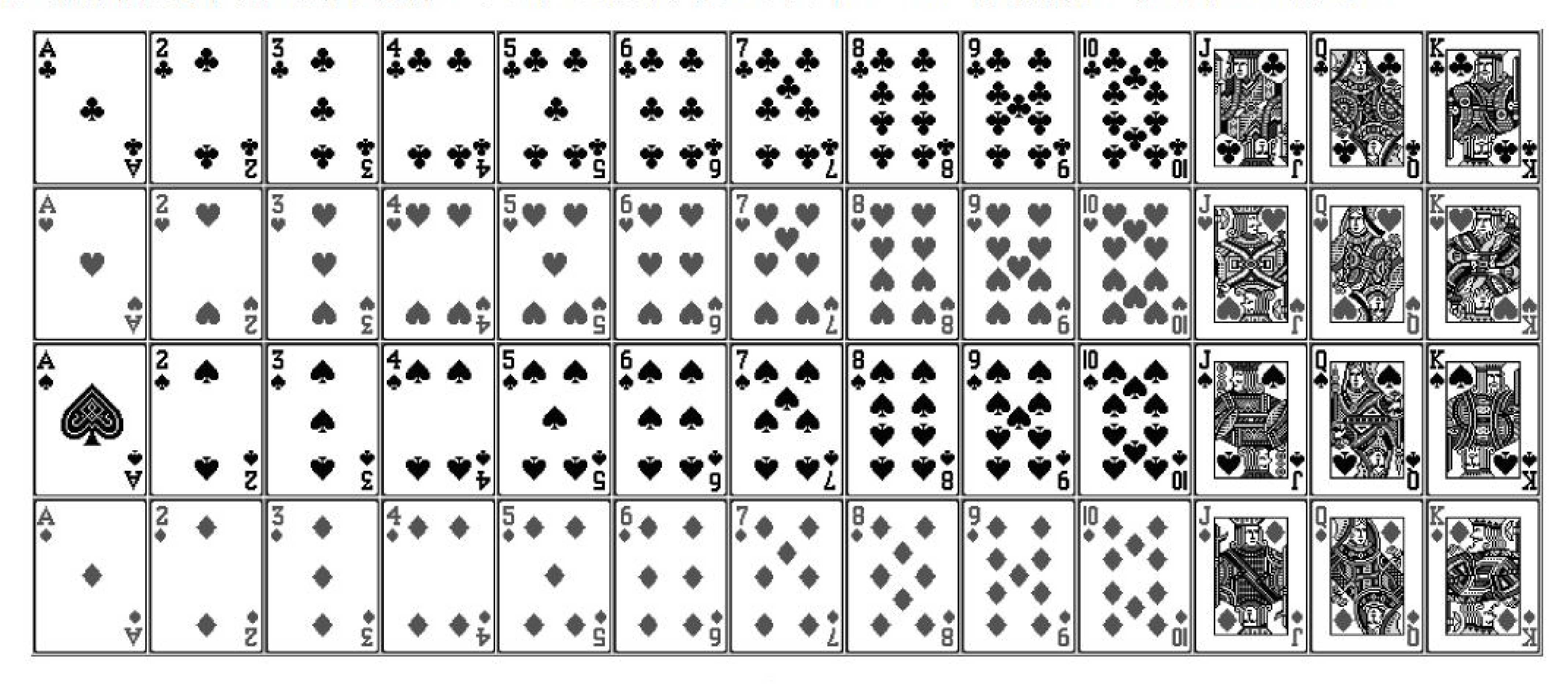


图 4.2



图像资源源自 https://mathematica. stackexchange.com/questions/16108/standard-deck-of-52-playing-cards-incurated-data。

当前目标是定义相关语法，根据花色确定纸牌，并按照下列方式排列：

```
val card = KING of HEARTS
```

首先，需要定义两种枚举值表示全部排列和花色，如下所示：

```
enum class Suit {
    HEARTS,
    SPADES,
    CLUBS,
    DIAMONDS
}

enum class Rank {
    TWO, THREE, FOUR, FIVE,
    SIX, SEVEN, EIGHT, NINE,
    TEN, JACK, QUEEN, KING, ACE;
}
```

随后，需要定义一个类，表示由排列和花色构成的纸牌，如下所示：

```
data class Card(val rank: Rank, val suit: Suit)
```

接下来，可按照下列方式实例化 Card 类：

```
val card = Card(Rank.KING, Suit.HEARTS)
```

为了进一步简化语法，可向 Rank 枚举值中引入新的中缀方法，如下所示：

```
enum class Rank {
    TWO, THREE, FOUR, FIVE,
    SIX, SEVEN, EIGHT, NINE,
    TEN, JACK, QUEEN, KING, ACE;

    infix fun of(suit: Suit) = Card(this, suit)
}
```

据此，可生成 Card 调用，如下所示：

```
val card = Rank.KING.of(Suit.HEARTS)
```

由于对应方法标记为 infix，因而可移除点号调用操作符和括号，如下所示：

```
val card = Rank.KING of Suit.HEARTS
```

静态导入的运用可进一步简化语法，进而实现最终目标，如下所示：

```
import Rank.KING
import Suit.HEARTS

val card = KING of HEARTS
```

除了简单之外，上述代码还具有 100%的类型安全性。也就是说，仅可通过预定义的 Rank 和 Suit 枚举值定义纸牌，且无法错误地确定某些不存在的纸牌。

4.13　可见性修饰符

Kotlin 支持 4 种类型的可见性修饰符（访问修饰符），即 private、protected、public 和 internal。Kotlin 并不支持包-private Java 修饰符。此处，二者间的主要差别在于，Kotlin 中，默认的可见性修饰符为 public，无须对其进行显式定义，因而对特定声明可予以忽略。全部修饰符均可应用于各种数据元素上，并可根据声明位置划分为两组，即顶级数据元素和嵌套成员。



在第 3 章中曾有所提及，顶级元素可直接声明于 Kotlin 文件中，这一点与嵌套于类、对象、接口和函数中的数据元素有所不同。在 Java 中，可在顶级仅声明类和接口；而 Kotlin 还可声明函数、对象、属性以及扩展。

下面首先讨论顶级数据元素的可见修饰符。

- public（默认状态）：数据元素于各处均处于可见状态。
- private：数据元素均在包含声明的文件内部处于可见状态。
- protected：无法在顶级位置获取。
- internal：数据元素在同一模块中处于可见状态。也就是说，在同一模块中，针对数据元素定义为 public。

什么是 Java 和 Kotlin 中的模块



模块是指共同参与编译的 Kotlin 文件集合，例如 IntelliJ IDEA 模块和 Gradle 项目。应用程序的模块化结构具备较好的发布机制，并可加速构建过程，其原因在于：仅重新编译发生变化的模块。

相关示例如下所示：

```
// top.kt
public val version: String = "3.5.0" // 1

internal class UnitConveter // 3
```

```
private fun printSomething() {
    println("Something")
}

fun main(args: Array<String>) {
    println(version) // 1, Prints: "3.5.0"
    UnitConveter() // 2, Accessible
    printSomething() // 3, Prints: Something
}

// branch.kt
fun main(args: Array<String>) {
    println(version) // 1, Accessible
    UnitConveter() // 2, Accessible
    printSomething() // 3, Error
}

// main.kt in another module
fun main(args: Array<String>) {
    println(version) // 1, Accessible
    UnitConveter() // 2, Error
    printSomething() // 3, Accessible
}
```

- 关于注释 1，version 属性定义为 public，因而在全部文件中均为有效。
- 关于注释 2，UnitConveter 可在 branch.kt 文件中予以访问。虽然位于同一模块，但 UnitConveter 并未处于 main.kt 文件中，而是位于另一个模块中。
- 关于注释 3，printSomething 函数仅在其所定义的同一文件中可被访问。

注意，Kotlin 中的包并不会产生任何额外的可见性权限。

第二组修饰符由成员构成，即声明于顶级元素内部的数据元素，一般是方法、属性、构造函数、对象（某些时候）、伴生对象、getter 和 setter，偶尔时候还包括嵌套类和嵌套接口。相关规则如下。

- public（默认状态）：查看声明类的客户端可看到 public 成员。
- private：数据元素仅在包含该成员的类或接口中可见。
- protected：在声明类和子类中可见，且在某个对象内部不具备可用性。
- internal：模块内的任意客户端（查看声明类）均可看到其内部成员。

下面定义顶级数据元素，该示例中将定义一个类，但同一逻辑可应用于包含嵌套成员的任意顶级数据元素上，如下所示：

```
class Person {
    public val name: String = "Igor"
```

```
    protected var age:Int = 23
    internal fun learn() {}
    private fun speak() {}
}
```

当生成 Person 类实例时，仅可访问标记为 public 修饰符的 name 属性，以及标记为 internal 修饰符的 learn 方法，如下所示：

```
// main.kt inside the same package as Person definition
val person = Person()
println(person.name)    // 1
person.speak()          // 2, Error
person.age              // 3, Error
person.learn()          // 4
```

- 对于注释 1，访问 Person 实例的客户端还可访问 name 属性。
- 对于注释 2，speak 方法仅在 Person 类内部可被访问。
- 对于注释 3，age 属性在 Person 类及其子类中可被访问。
- 针对注释 4，位于模块内部且访问 Person 类实例的客户端还可访问其 public 成员。

继承可访问性类似于外部可访问性，二者间的主要差别在于，标记为 protected 修饰符的成员在子类中也处于可见状态，如下所示：

```
open class Person {
    public val name: String = "Igor"
    private fun speak() {}
    protected var age: Int = 23
    internal fun learn() {}
}

class Student() : Person() {
    fun doSth() {
        println(name)
        learn()
        print(age)
        // speak() // 1
    }
}
```

在 Student 子类中，可访问标记为 public、protected、internal 的成员，但不包括标记为 private 修饰符的成员。

public、private 和 protected 修饰符在 Java 中均包含对等的修饰符，但并不包括 internal，因而不支持 Java 中的字节码。这也是 internal 修饰符直接编译为 public 的原因。当与其通

信时将无法在 Java 中使用——其名称已经发生变化，且无法再次使用。例如，当定义下列 Foo 类时：

```
open class Foo {
    internal fun boo() { }
}
```

可通过 Java 并按照如下方式对其加以使用：

```
public class Java {
    void a() {
        new Foo().boo$production sources for module SmallTest();
    }
}
```

具有争议的是，内部可见性由 Kotlin 保护，并可以通过使用 Java 适配器避开这一方式，除此之外，不存在其他实现方式。

除了在类中定义可见性修饰符之外，还可在重载某个成员时对修饰符进行重载，从而可弱化继承体系结构中的访问限制条件，如下所示：

```
open class Person {
    protected open fun speak() {}
}

class Student() : Person() {
    public override fun speak() {
    }
}

val person = Person()
// person.speak() // 1

val student = Student()
student.speak() // 2
```

- 针对注释 1，鉴于 protected 特性，无法访问 speak 方法。
- 针对注释 2，speak 方法的可见性调整为 public，因而可对其进行访问。

针对成员及其可见性范围，修饰符的定义方式较为直观。下面考察类和构造函数可见性的定义方式。如前所述，主构造函数定义位于类头部位置，因而一行代码中需要定义两个可见性修饰符，如下所示：

```
internal class Fruit private constructor {
   var weight: Double? = null
```

```
    companion object {
        fun create() = Fruit()
    }
}
```

假设上述类定义在顶部（顶级），因而在当前模块内可见，但仅可从包含类声明的文件中予以实例化，如下所示：

```
var fruit: Fruit? = null // Accessible
fruit = Fruit()          // Error
fruit = Fruit.create()   // Accessible
```

默认条件下，getter 和 setter 与属性具有相同的可见性修饰符，但也可对其进行修改。相应地，Kotlin 可在 get/set 关键字之前设置可见性修饰符，如下所示：

```
class Car {
    init {
        count++;
        println("Car created")
    }

    companion object {
        init {
            println("Car companion object created")
        }
        var count: Int = 0
            private set
    }
}
```

在上述示例中，getter 的可见性已被调整。需要注意的是，该方案可在不改变（编译器生成的）默认实现的前提下修改可见性修饰符。作为只读型外部客户端，实例计数器当前处于安全状态，但仍可从 Car 类内部调整属性。

4.14 密 封 类

密封类包含了有限数量的子类（密封的子类型层次结构）。在 Kotlin 1.1 之前，子类须在密封类体中定义。Kotlin 1.1 弱化了这一限制条件，并可作为密封类声明在同一文件中定义密封类子类。其中，全部类均彼此靠近而声明，因而可简单地查看一个文件，即可方便地看到全部子类，如下所示：

```
// vehicle.kt

sealed class Vehicle()
class Car : Vehicle()
class Truck : Vehicle()
class Bus : Vehicle()
```

当标记密封类时，可向类声明添加 sealed 修饰符。上述声明表示，Vehicle 类仅可通过 3 个类被扩展，即 Car、Truck 以及 Bus 类，且全部声明于同一个文件中。除此之外，还可在 vehicle.kt 文件中添加第 4 个类，但却无法在另一个文件中定义该类。

密封子类型限制仅适用于 Vehicle 类的直接继承者，这意味着，Vehicle 仅可被定义于相同文件中的类扩展（Car、Truck 或 Bus）。假设 Car、Truck 或 Bus 类处于 open 状态，随后可通过声明于任意文件内部的某个类进行扩展，如下所示：

```
// vehicle.kt
sealed class Vehicle()
open class Bus : Vehicle()

// data.kt
class SchoolBus:Bus()
```

为了避免这一行为，需要将 Car、Truck 或 Bus 标记为 sealed，如下所示：

```
// vehicle.kt
sealed class Vehicle()
sealed class Bus : Vehicle()

// data.kt
class SchoolBus:Bus() // Error cannot access Bus
```

密封类可与 when 表达式实现良好的协同工作。由于编译器可验证：密封类的各个子类在 when 代码块中包含对应的子句，因而无须设置 else 子句，如下所示：

```
when (vehicle) {
    is Car -> println("Can transport 4 people")
    is Truck -> println("Can transport furnitures ")
    is Bus -> println("Can transport 50 people ")
}
```

此处可安全地向 Vehicle 类添加新的子类——如果在程序某处缺失对应的 when 表达式子句，该程序将无法编译。这也修复了 Java 中的 switch 语句，其中，程序员往往会忘记添加相关内容，这将导致程序崩溃，或者产生未知错误。

默认状态下，密封类为抽象类，因而无须添加 abstract 修饰符。另外，密封类不可定

义为 open 或 final 状态。同时，当需要确保仅存在单一实例时，可利用对象替换子类，如下所示：

```
sealed class Employee()

class Programmer : Employee()
class Manager : Employee()
object CEO : Employee()
```

上述声明不仅保护了继承层次结构，同时还将 CEO 限定为单一实例。关于密封类，更为详细的讨论已超出了本书的范围，但读者须注意以下几点内容：

- 定义诸如链表或二叉树数据类型（读者可参考 https://en.wikipedia.org/wiki/Algebraic_data_ type）。
- 当构造应用程序模块或库时，若不允许客户端扩展类，但依然可亲自对其进行扩展时，应保护继承层次结构。
- 在状态机中，某些状态可能会包含其他状态中的无意义数据（参考 https://en.wikipedia. org/ wiki/Finite- state_ machine）。
- 词法分析中的标记类型列表。

4.15 嵌 套 类

嵌套类表示为定义于另一个类中的数据类。在顶级类中嵌套小型数据类可将代码置于所用附近处，并以较好的方式对类进行组织。对此，典型的示例是 Tree/Leaf 监听器或状态表述。类似于 Java，Kotlin 也可定义嵌套类，并存在两种处理方式。例如，可将类作为成员在某个类中加以定义，如下所示：

```
class Outer {
    private val bar: Int = 1

    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2
```

上述示例可生成 Nested 类实例，且无须创建 Outer 类实例。其中，类无法直接引用实例变量或定义于其封闭类中的相关方法（仅可通过对象引用加以使用）。这等价于 Java 中的静态嵌套类以及静态成员。

当访问外部类的成员时，需要将嵌套类标记为 inner，进而创建第二类数据类，如下所示：

```
class Outer {
    private val bar: Int = 1

    inner class Inner {
        fun foo() = bar
    }
}

val outer = Outer()
val demo = outer.Inner().foo() // == 1
```

当实例化 inner 类时，首先需要实例化 Outer 类。此处，Inner 类可访问定义于外部类中的全部方法和属性，并与外部类共享状态，每个 Outer 类中仅可存在单一的 Inner 类实例。表 4.5 总结了其中的差异。

表 4.5

行为	类（成员）	内部类（成员）
类似于 Java 中的静态成员	是	否
该类实例可在缺少封闭类实例情况下存在	是	否
包含外部类的引用	否	是
与外部类共享状态（可访问外部类成员）	否	是
实例的数量	无限制	每个外部类实例包含 1 个实例

当决定是否定义 inner 类或顶级类时，应考察潜在的类应用方式。如果类仅对单一类实例有效，则应将其声明为 inner；如果与外部类相比，内部类在另一个上下文环境中更为有效，则须将其声明为顶级类。

4.16　导 入 别 名

别名是一种新的类型名称引入方式。如果类型名称已用于某个文件中，但该名称可能不太适宜或者较长，对此，可引入不同的名称加以使用，而不再使用原类型名称。注意，别名并未引入新的类型，且仅在编译期之前有效（编写代码时）。最终，编译器将利用实际类替换类别名，因而别名在运行期内不复存在。

某些时候，需要在单一文件中使用包含相同名称的多个类。例如，InterstitialAd 类型分别定义于 Facebook 和 Google 广告库中，假设在某个单一文件中均需要使用到这两个库。这在项目开发中较为常见，以在两个广告供应商之间实现效益对比。此处的问题是，在单一文件中使用两种数据类型则意味着，须通过完全限定类名（命名空间+类名）对其进行访问，如下所示：

```
import com.facebook.ads.InterstitialAd

val fbAd = InterstitialAd(context, "...")
val googleAd = com.google.android.gms.ads.InterstitialAd(context)
```

限定类名和非限定类名



非限定类名简单地表示为类名，例如 Box；而限定类名则表示为命名空间和类名的结合体，例如 com.test.Box。

在这一类使用环境中，可能有人认为较好的做法是重命名某个类。但某些场合下，该方法并不可行（对应类定义于外部库中），或者并非是一种期望行为（类名与后台数据库表保持一致）。对此，上述两个类均位于外部库中，类命名冲突的解决方法是使用 import 别名。相应地，可将 Google InterstitialAd 重命名为 GoogleAd，并将 Facebook InterstitialAd 重命名为 FbAd，如下所示：

```
import com.facebook.ads.InterstitialAd as FbAd
import com.google.android.gms.ads.InterstitialAd as GoogleAd
```

当前，可像实际类型那样针对文件使用别名，如下所示：

```
val fbAd = FbAd(context, "...")
val googleAd = GoogleAd(context)
```

通过 import 别名，可显式地重定义导入至某一文件中的类名。在这种情况下，无须使用两个别名，但这将会提升代码的可读性——与 InterstitialAd 和 GoogleAd 相比，较好的方法是设置 FbAd 和 GoogleAd。由于可简单地告知编译器：每次遇到 GoogleAd 别名时，可在编译过程中将其转换为 com.google.android.gms.ads.InterstitialAd；每次遇到 FbAdalias 时可将其转换为 com.facebook.ads.InterstitialAd，因而无须再使用完全限定类名。另外，导入 birmingham 仅工作于别名所定义的文件内。

4.17　本 章 小 结

本章讨论了针对面向对象程序设计的构造块，如何定义接口、类以及 inner、sealed、

enum 和数据类之间的差异。其中，全部元素在默认条件下均为 public 类型，且全部类/接口均为 final 类型（默认状态下），因而需要显式地执行“开启”操作，进而支持继承和成员重载机制。

本章还介绍了如何通过简明的数据类（与功能强大的属性相结合）定义相应的数据模块；如何利用编译器生成的方法对数据进行操作，以及如何重载操作符。

另外，本章还探讨了如何通过对象声明生成单例；如何定义匿名类型对象，并以此通过对象表达式扩展某些类和/或实现某些接口。除此之外，本章还展示了 lateinit 修饰符的具体应用，进而利用延迟初始化行为定义非空数据类型。

第 5 章将考察 Kotlin 中的函数，并讨论与函数式编程（FP）相关的概念，包括函数类型、lambdas 以及高阶函数。

第 5 章　函数——一等公民

第 4 章介绍了 Kotlin 中与 OOP 相关的特性。本章则主要讨论之前标准 Android 开发中未涉及的高级函数特性。其中，某些特性在 Java 8 中被引入（在 Android 中通过 Retrolambda 插件），但 Kotlin 中涵盖了大量的函数编程特性。

本章主要介绍高级函数，并将其视为“一等公民”。大多数概念与读者之前所见的函数型语言并无太多差异。

本章主要涉及以下内容：

- 函数类型。
- 匿名函数。
- Lambda 表达式。
- Lambda 表达式中独立参数的隐式名称。
- 高阶函数。
- 参数规则中最后一个 Lambda。
- Java 单一抽象方法（SAM）接口。
- 参数应用中，基于 Java 单一抽象方法的 Java 方法。
- 函数类型中的命名参数。
- 类型别名。
- 内联函数。
- 函数引用。

5.1　函数类型

Kotlin 支持函数编程，在 Kotlin 中，函数被视为“一等公民”。在现有的编程语言中，术语“一等公民”描述了一类实体，并支持其他实体中可用的全部操作。此类操作一般包括参数传递、函数中的返回机制以及变量赋值。因此，“Kotlin 中函数视作一等公民”应该理解为：在 Kotlin 中，可作为参数传递函数；可从函数中返回函数；将函数赋予某个变量中。虽然 Kotlin 是一类静态类型语言，但仍须定义某种函数类型，以支持各项操作。在 Kotlin 中，定义函数类型的符号如下所示：

```
(types of parameters)->return type
```

相关示例如下所示：

- (Int)->Int：函数接收 Int 作为参数，并返回 Int。
- ()->Int：函数未接收任何参数，并返回 Int。
- (Int)->Unit：函数接收 Int 参数，且不返回任何内容（仅为 Unit，且无须返回）。

下列内容显示了可载入函数的属性示例：

```
lateinit var a: (Int) -> Int
lateinit var b: ()->Int
lateinit var c: (String)->Unit
```



术语“函数类型”通常定义为赋予函数的变量或参数类型，或者是接收或返回某个函数的、高阶函数的参数或结果类型。在 Kotlin 中，函数类型可像接口那样被处理。

本章后续内容将会看到，Kotlin 函数可在参数中使用其他函数，甚至可返回函数，如下所示：

```
fun addCache(function: (Int) -> Int): (Int) -> Int {
    // code
}

val fibonacciNumber: (Int)->Int = // function implementation
val fibonacciNumberWithCache = addCache(fibonacciNumber)
```

如果某个函数可接收或返回函数，那么，该函数类型应能够定义相关函数，并以参数形式接收函数，或者返回某个函数。对此，可简单地将函数类型符号定义为一个参数或返回类型。相关示例如下所示。

- (String)->(Int)->Int：函数接收 String，并返回一个函数（接收 Int 类型并返回 Int）。
- (()->Int)->String：函数接收另一个函数作为参数，并返回 String 类型。其中，参数中的函数并不接收任何参数，且返回 Int。

基于函数的各项属性可像某个函数那样被调用，如下所示：

```
val i = a(10)
val j = b()
c("Some String")
```

函数不仅可存储于变量中，还可用作泛型。例如，可将函数保存至列表中，如下所示：

```
var todoList: List<() -> Unit> = // ...
for (task in todoList) task()
```

上述列表可存储包含() -> Unit 签名的函数。

实际上，函数类型仅表示为针对通用接口的一类语法糖。下面考察某些示例：

- ()->Unit 签名表示为针对 Function0<Unit>的一个接口。由于包含了 0 个参数，因而表达式为 Function0，且返回类型为 Uint。
- (Int)->Unit 签名表示为针对 Function1<Int, Unit>的接口。由于包含了 1 个参数，因而表达式为 Function1。
- ()->(Int, Int)->String 签名表示针对 Function0<Function2<Int, Int, String>>的接口。

上述接口仅包含一个方法 invoke，同时表示为一个操作符，并支持某个对象可像函数那样加以使用，如下所示：

```
val a: (Int) -> Unit = //...
a(10)        // 1
a.invoke(10)// 1
```

针对注释 1，两条语句具有相同的含义。

函数接口并未在标准库中体现，并可视作整合后的、编译器生成的类型（在编译期间生成）。据此，在函数类型参数的数量上并不存在人为的限制，且标准库的尺寸也不会增加。

5.2 匿名函数

一种将函数定义为对象的方式是使用匿名函数，其工作方式与常规函数相同，但在 fun 关键字和参数声明之间并不包含名称。因此，默认条件下将被视为对象。对应示例如下所示：

```
val a: (Int) -> Int = fun(i: Int) = i * 2 // 1
val b: ()->Int = fun(): Int { return 4 }
val c: (String)->Unit = fun(s: String){ println(s) }
```

对于注释 1，代码表示为匿名单一表达式函数。需要注意的是，类似于常规独立表达式函数，当从表达式返回类型进行推断时，返回类型无须被指定。

考察下列应用示例：

```
// Usage
println(a(10))      // Prints: 20
println(b())        // Prints: 4
c("Kotlin rules")   // Prints: Kotlin rules
```

在上述示例中，函数类型采用显式方式加以定义；而 Kotlin 中包含了较好的类型推断机制，函数类型可从匿名默认函数定义的类型进行推断，如下所示：

```
var a = fun(i: Int) = i * 2
var b = fun(): Int { return 4 }
var c = fun(s: String){ println(s) }
```

除此之外，还可按照相反的方式工作。当定义某个属性的类型时，考虑到推断机制，无须在匿名函数中显式地设置参数类型，如下所示：

```
var a: (Int)->Int = fun(i) = i * 2
var c: (String)->Unit = fun(s){ println(s) }
```

当检测函数类型的方法时，将会看到其中仅存在 invoke 方法。这里，invoke 方法定义为一个操作符函数，其应用方式与函数调用相同。因此，在括号内使用 invoke 调用将得到相同的结果，如下所示：

```
println(a.invoke(4))        // Prints: 8
println(b.invoke())         // Prints: 4
c.invoke("Hello, World!")   // Prints: Hello, World!
```

这一点十分重要，例如在某个可空变量中设置函数时。对此，可通过安全调用使用 invoke 方法，如下所示：

```
var a: ((Int) -> Int)? = null // 1
if (false) a = fun(i: Int) = i * 2
print(a?.invoke(4)) // Prints: null
```

在注释 1 中，a 可空，因而通过安全调用使用 invoke 方法。

下面考察一个 Android 示例。通常情况下，需要定义一个独立的错误处理程序，其中包含了多个日志方法，并作为参数将其传递至不同的对象中。下列代码展示了如何通过匿名函数对其加以实现。

```
val TAG = "MainActivity"
val errorHandler = fun (error: Throwable) {
    if(BuildConfig.DEBUG) {
        Log.e(TAG, error.message, error)
   }
   toast(error.message)
   // Other methods, like: Crashlytics.logException(error)
}

// Usage in project
val adController = AdController(errorHandler)
val presenter = MainPresenter(errorHandler)
```

```
// Usage
val error = Error("ExampleError")
errorHandler(error) // Logs: MainActivity: ExampleError
```

匿名函数简单而有效，并可定义函数以作为对象加以使用和传递。除此之外，还存在更为简单的方式可实现类似的行为，即 Lambda 表达式。

5.3　Lambda 表达式

在 Kotlin 中，最为简单的匿名函数定义方式是使用 Lambda 表达式。该表达式与 Java 8 中的 Lambda 表达式类似，二者间的主要差别在于，Kotlin 中的 Lambda 表达式实际上为闭包，因而可根据构建上下文修改变量。这在 Java 8 中并不可行，本章稍后将探讨其中的差异。下面考察一些简单的示例。Kotlin 中的 Lambda 表达式包含下列符号：

```
{ arguments -> function body }
```

其中，将返回最后一个表达式的结果。下面是一些简单的 Lambda 表达式示例。

- { 1 }：该 Lambda 表达式不包含任何参数并返回 1，其类型为()->Int。
- { s: String -> println(s) }：该 Lambda 表达式接收类型为 String 的 1 个参数并对其予以输出；同时返回 Unit，对应类型为(String)->Unit。
- { a: Int, b: Int -> a + b }：该 Lambda 表达式接收两个 Int 参数，并返回二者之和，对应类型为(Int, Int)->Int。

在第 3 章中，所定义的函数可采用 Lambda 表达式加以定义，如下所示：

```
var a: (Int) -> Int = { i: Int -> i * 2 }
var b: ()->Int = { 4 }
var c: (String)->Unit = { s: String -> println(s) }
```

返回类型源自 Lambda 表达式中的最后一条语句，但除非包含了某个标记限定的 return 语句，否则，返回操作行为并不被允许，如下所示：

```
var a: (Int) -> Int = { i: Int -> return i * 2 }
// Error: Return is not allowed there
var l: (Int) -> Int = l@ { i: Int -> return@l i * 2 }
```

另外，Lambda 还可包含多行内容，如下所示：

```
val printAndReturn = { i: Int, j: Int ->
    println("I calculate $i + $j")
    i + j // 1
}
```

针对注释 1，该语句为最后一条语句，因而该表达式的结果将表示为返回值。

当采用分号进行分割时，多行语句也可在一行中加以定义，如下所示：

```
val printAndReturn = {i: Int, j: Int -> println("I calculate $i + $j");
                      i + j }
```

Lambda 表达式不仅可在参数提供的数值上进行操作，还可根据构建上下文使用全部属性和函数，如下所示：

```
val text = "Text"
var a: () -> Unit = { println(text) }
a() // Prints: Text
a() // Prints: Text
```

这也是 Kotlin 和 Java 8 之间的最大差异。Java 匿名对象和 Java 8 Lambda 表达式可使用源自上下文中的字段，但 Java 并不允许向此类变量赋予不同的值（用于 Lambda 中的 Java 变量须为 final，如图 5.1 所示）。

```
import kotlin.jvm.functions.Function0;

public class JavaTest {
    public static void main(String[] args) {
        int counter = 0;
        Function0<Void> func = () -> {
            counter++;
Variable used in lambda expression should be final or effectively final
    }
}
```

图 5.1

Kotlin 在这一方面则更胜一筹：Lambda 表达式和匿名函数可修改上述值。封装了局部变量并可在函数体内对其进行修改的 Lambda 表达式称作闭包。Kotlin 支持完整的闭包定义。为了避免 Lambda 和闭包之间的混淆，本书将二者统称为 Lambda。考察下列示例：

```
var i = 1
val a: () -> Int = { ++i }
println (i)      // Prints: 1
println (a())    // Prints: 2
println (i)      // Prints: 2
println (a())    // Prints: 3
println (i)      // Prints: 3
```

Lambda 表达式可用于修改局部上下文环境中的变量，下列代码展示了一个计数器示例，其中，对应数值保存于局部变量中，如下所示：

```
fun setUpCounter() {
    var value: Int = 0
    val showValue = { counterView.text = "$value" }
    counterIncView.setOnClickListener { value++; showValue() }
    // 1
    counterDecView.setOnClickListener { value--; showValue() }
    // 1
}
```

针对注释 1，通过 Lambda 表达式可在 Kotlin 中设置 View 的 onClickListener。

需要注意的是，在上述示例中，并未指定 showValue 类型，其原因在于：在 Kotlin 的 Lambda 表达式中，若编译器可从上下文中进行推断时，类型参数为可选类型，如下所示：

```
val a: (Int) -> Int = { i -> i * 2 } // 1
val c: (String)->Unit = { s -> println(s) } // 2
```

针对注释 1，i 的推断类型为 Int——该函数类型定义为 Int 参数；对于注释 2，s 的推断类型表示为 String——函数类型定义为 String 参数。

在后续示例中将会看到，由于可根据属性类型进行推断，因而无须指定参数类型，除此之外，类型推断还可以另一种方式工作：可定义 Lambda 表达式参数的类型，进而推断属性类型，如下所示：

```
val b = { 4 }                          // 1
val c = { s: String -> println(s) }    // 2
val a = { i: Int -> i * 2 }            // 3
```

针对注释 1，由于 4 为 Int 且不存在参数类型，因而推断类型为 type is ()->Int。对于注释 2，由于参数类型为 String，且 println 方法的返回类型表示为 Unit，因而推断类型为(String)->Unit。对于注释 3，由于 i 为 Int，且 Int 的乘法运算操作的返回类型为 Int，因而推断类型为(Int)->Int。

该推断简化了 Lambda 表达式的定义。通常情况下，当把 Lambda 表达式定义为函数参数时，无须每次指定参数类型。除此之外，当参数类型可被推断时，可针对单一参数 Lambda 表达式使用更为简单的标记，稍后将对此加以讨论。

当满足下列两个条件时，可忽略 Lambda 参数定义，并使用 it 关键字访问参数：

- 仅存在一个参数。
- 参数类型可根据上下文进行推断。

作为示例，下面再次定义属性 a 和 c，但使用单一参数的隐式名称，如下所示：

```
val a: (Int) -> Int = { it * 2 }            // 1
val c: (String)->Unit = { println(it) }     // 2
```

针对注释 1，代码等同于{ i -> i * 2 }；针对注释 2，代码等同于{ s -> println(s) }。

上述符号在 Kotlin 中十分常见，其优点体现在短小精悍，同时还可避免参数定义。除此之外，还提升了定义于 LINQ 风格中处理过程的可读性。该风格需要使用到某些尚未被引入的组件，此处仅为展示这一概念。考察下列示例：

```
strings.filter { it.length = 5 }.map { it.toUpperCase() }
```

其中，假设字符串表示为 List<String>，该表达式利用“长度等于 5”这一条件过滤字符串，并将其转换为大写形式。

需要注意的是，在 Lambda 表达式体中，可使用 String 类中的相关方法，其原因在于：函数类型（例如针对 filter 的(String)->Boolean）根据当前方法定义进行推断，即根据可迭代类型（List<String>）推断 String。另外，返回列表类型（List<String>）取决于 Lambda（String）返回的内容。

LINQ 风格在函数式语言中较为常见，并使得集合语法或者 String 的处理过程更为简洁。第 7 章将对此加以深入介绍。

5.4　高阶函数

高阶函数是指，函数至少接收一个函数作为参数，或者返回一个函数作为最终结果。Kotlin 对此提供了完整的支持。此处假设需要定义两个函数，函数一将从列表中添加全部 BigDecimal 数字；函数二将获取全部数字的乘积结果（列表中全部元素间的乘法运算结果），如下所示：

```
fun sum(numbers: List<BigDecimal>): BigDecimal {
    var sum = BigDecimal.ZERO
    for (num in numbers) {
        sum += num
    }
    return sum
}

fun prod(numbers: List<BigDecimal>): BigDecimal {
    var prod = BigDecimal.ONE
    for (num in numbers) {
        prod *= num
    }
    return prod
}
```

```
// Usage
val numbers = listOf(
    BigDecimal.TEN,
    BigDecimal.ONE,
    BigDecimal.valueOf(2)
)
print(numbers)             //[10, 1, 2]
println(prod(numbers))    // 20
println(sum(numbers))     // 13
```

上述函数基本相同，唯一的差别在于名称、累加器（BigDecimal.ZERO 或 BigDecimal.ONE）以及相关操作。当采用 DRY（不要重复自己）规则时，则不应保留项目中的两部分类似代码。虽然定义一个具有相似行为，但在使用的对象中却有不同的函数很容易；相比之下，定义一个执行操作中有所差异的函数却较为困难（此处，函数的差异体现在累积过程中的相关操作）。由于可将当前操作作为参数加以传递，因而函数类型可视为一种解决方案。在该示例中，可通过下列方式获取常用方法：

```
fun sum(numbers: List<BigDecimal>) =
    fold(numbers, BigDecimal.ZERO) { acc, num -> acc + num }

fun prod(numbers: List<BigDecimal>) =
   fold(numbers, BigDecimal.ONE) { acc, num -> acc * num }

private fun fold(
    numbers: List<BigDecimal>,
    start: BigDecimal,
    accumulator: (BigDecimal, BigDecimal) -> BigDecimal
): BigDecimal {
    var acc = start
    for (num in numbers) {
        acc = accumulator(acc, num)
    }
    return acc
}

// Usage

fun BD(i: Long) = BigDecimal.valueOf(i)
val numbers = listOf(BD(1), BD(2), BD(3), BD(4))
println(sum(numbers))  // Prints: 10
println(prod(numbers)) // Prints: 24
```

其中，fold 函数遍历数字，并利用各项元素更新 acc。需要注意的是，函数参数的定义类似于其他类型，并可像其他函数那样加以使用。例如，可定义 vararg 函数类型参数，如下所示：

```
fun longOperation(vararg observers: ()->Unit) {
    //...
    for(o in observers) o()
}
```

在 longOperation 中，for 用于遍历全部观察者，并依次对其加以调用。该函数支持多个函数作为参数，相关示例如下所示：

```
longOperation({ notifyMainView() }, { notifyFooterView() })
```

另外，Kotlin 中的函数还可返回函数。例如，可定义某个函数，并创建自定义错误处理程序，其中包含了相同的错误日志机制，以及不同的标签，如下所示：

```
fun makeErrorHandler(tag: String) = fun (error: Throwable) {
    if(BuildConfig.DEBUG) Log.e(tag, error.message, error)
    toast(error.message)
    // Other methods, like: Crashlytics.logException(error)
}

// Usage in project
val adController = AdController(makeErrorHandler("Ad in MainActivity"))

val presenter = MainPresenter(makeErrorHandler("MainPresenter"))

// Usage
val exampleHandler = makeErrorHandler("Example Handler")
exampleHandler(Error("Some Error")) // Logs: Example Handler: Some
Error
```

当采用参数中的函数时，较为常见的 3 种情形包括：

- 向函数提供操作。
- 观察者（监听器）模式。
- 线程操作后的回调行为。

下面对此予以详细介绍。

5.4.1　向函数提供操作

第 4 章曾有所提及，某些时候，需要从函数中获取常用功能项，但其操作却有所差异。

在这种情况下，我们仍可获得该功能项，但需要提供一个参数，并包含了对其予以区分的操作。通过这种方式，即可获取并复用任意常见模式。例如，通常仅需要与某些断言相匹配的列表元素，如仅需要显示某些活动元素。对此，较为经典的实现方法如下所示：

```
var visibleTasks = emptyList<Task>()
for (task in tasks) {
    if (task.active)
    visibleTasks += task
}
```

尽管这可视作一类常见操作，但仍可根据当前断言（predicate）获取功能项（仅过滤某些元素），进而分离函数，同时更加方便地对其加以使用，如下所示：

```
fun <T> filter(list: List<T>, predicate: (T)->Boolean) {
    var visibleTasks = emptyList<T>()
    for (elem in list) {
        if (predicate(elem))
            visibleTasks += elem
    }
}

var visibleTasks = filter(tasks, { it.active })
```

这种高阶函数的应用方式十分重要，本书将通篇对此加以描述，但并非是高阶函数唯一的应用方式。

5.4.2　观察者（监听器）模式

当引发某一事件并执行相关操作时，可采用 Observer（Listener）模式。在 Android 开发中，观察者常被设置为视图元素。常见的示例包括单击监听器、触摸监听器或文本查看器。在 Kotlin 中，可设置不包含任何样本文件的监听器。例如，下列代码显示了按钮单击操作的监听器：

```
button.setOnClickListener({ someOperation() })
```

注意，setOnClickListener 表示为 Android 库中的 Java 方法。稍后将会详细解释为何要利用 Lambda 表达式对其加以使用。监听器的构建过程较为简单，相关示例如下所示：

```
var listeners: List<()->Unit> = emptyList() // 1
fun addListener(listener: ()->Unit) {
    listeners += listener // 2
}
```

```
fun invokeListeners() {
    for( listener in listeners) listener() // 3
}
```

针对注释 1，创建一个空列表并加载全部监听器；对于注释 2，可简单地将某个监听器添加至监听器列表中；在注释 3 中，遍历监听器并逐一对其加以调用。

该模式的实现过程十分简单。除此之外，另一种常见应用（基于函数类型的参数应用）是线程操作后的回调。

5.4.3　线程操作后的回调

如果操作较为耗时，且不希望用户长时间等待，则需要在另一个线程中启动该操作。为了在独立线程中调用耗时操作之后使用回调，可将其作为参数予以传递，示例函数如下所示：

```
fun longOperationAsync(longOperation: ()->Unit, callback: ()->Unit) {
        Thread({ // 1
            longOperation() // 2
        callback() // 3
    }).start() // 4
}

// Usage
longOperationAsync(
        longOperation = { Thread.sleep(1000L) },
        callback = { print("After second") }
        // 5, Prints: After second
)
println("Now") // 6, Prints: Now
```

在注释 1 中，创建了 Thread，同时还传递了希望在构造器参数上执行的 Lambda 表达式；在注释 2 中，执行了较为耗时的操作；在注释 3 中，启用了当前参数提供的回调操作；在注释 4 中，start 表示为启动所定义线程的方法；在注释 5 中，1 秒延迟后实现输出；在注释 6 中，实现了即时输出。

实际上，对于回调应用，还存在一些较为常见的替代方法，例如 RxJava。当然，经典的回调方式依然十分常见。在 Kotlin 中，其实现方式一般不包含任何样板文件。

当采用高阶函数时，上述形式是最为常见的用例，旨在获取常见行为并减少样板文件。关于高阶函数，Kotlin 支持大量的改进措施。

5.5　命名参数和 Lambda 表达式的组合

在 Android 开发中，默认命名参数和 Lambda 表达式十分有用，下面考察 Android 中的一些实际用例。假设通过某个函数下载某些元素，并向用户显示，其中将添加下列参数：

- onStart：在网络开启之前进行调用。
- onFinish：在网络开启之后被调用。

```
fun getAndFillList(onStart: () -> Unit = {},
    onFinish: () -> Unit = {}){
        // code
}
```

随后，即可在 onStart 和 onFinish 中显示或隐藏旋转加载图标，如下所示：

```
getAndFillList(
    onStart = { view.loadingProgress = true } ,
    onFinish = { view.loadingProgress = false }
)
```

如果需要从 swipeRefresh 启动，仅须在其结束时进行隐藏，如下所示：

```
getAndFillList(onFinish = { view.swipeRefresh.isRefreshing =
   false })
```

如果需要执行更新操作，则可按照下列方式加以调用：

```
getAndFillList()
```

对于多功能函数来讲，命名参数和 Lambda 表达式可视为一种较为完美的组合，可选取期望实现的参数，以及应实现的相关操作。如果某个函数包含了多个函数类型参数，那么，在大多数时候，应通过命名参数语法加以使用，其原因在于，当使用多个函数类型参数时，Lambda 表达式一般不具备自解释特性。

5.6　参数规则中最后一个 Lambda

在 Kotlin 中，高阶函数十分重要，这也是 Kotlin 对此引入了特殊规则的原因——使高阶函数更加简单明了，其工作方式可描述为：如果最后一个参数表示为函数，则可在括号外部定义一个 Lambda 表达式。下面考察与 longOperationAsync 结合使用时，该表达式的表现结果，相关定义如下所示：

```
fun longOperationAsync(a: Int, callback: ()->Unit) {
    // ...
}
```

该函数类型位于参数中的最后位置，并可通过下列方式执行：

```
longOperationAsync(10) {
    hideProgress()
}
```

考虑到参数规则中最后一个 Lambda，可将 Lambda 置于括号之后，以使其看起来位于参数外部。

作为示例，下面考察在 Kotlin 中，代码在另一个线程中的调用方式。在 Kotlin 中，新线程的标注启动方式可描述为：从 Kotlin 标准库中使用线程函数，其定义如下所示：

```
public fun thread(
    start: Boolean = true,
    isDaemon: Boolean = false,
    contextClassLoader: ClassLoader? = null,
    name: String? = null,
    priority: Int = -1,
    block: () -> Unit): Thread {
        // implementation
    }
```

不难发现，block 参数接收异步调用的操作，并位于最后一个位置。其他参数包含默认的定义参数。thread 函数的使用方式如下所示：

```
thread { /* code */ }
```

thread 定义包含了多个其他参数，其设置方式可使用命名参数语法，或者采用逐一方式予以提供，如下所示：

```
thread (name = "SomeThread") { /*...*/ }
thread (false, false) { /*...*/ }
```

参数规则中的最后一个 Lambda 表示为语法糖，但可简化高阶函数的应用方式。该规则的差异性主要体现在下列两个常见用例中：

- 命名代码的包围机制。
- 利用 LINQ 风格处理数据结构。

下面对此加以详细讨论。

5.6.1　命名代码的包围机制

某些时候，需要以不同方式标记执行代码中的部分内容，thread 函数即属于这一类情况。其中，代码须异步执行，因而可采用起始于 thread 函数的括号对其加以包围，如下所示：

```
thread {
    operation1()
    operation2()
}
```

从外部来看，这似乎是代码的一部分内容，并被名为 thread 的代码块所围绕。下面考察另一个示例，并假设需要记录特定代码块的执行时间。作为一个帮助函数，可定义 addLogs 函数，该函数输出日志和时间，其定义方式如下所示：

```
fun addLogs(tag: String, f: () -> Unit) {
    println("$tag started")
    val startTime = System.currentTimeMillis()
    f()
    val endTime = System.currentTimeMillis()
    println("$tag finished. It took " + (endTime - startTime))
}
```

该函数的应用方式如下所示：

```
addLogs("Some operations") {
    // Operations we are measuring
}
```

对应的执行示例如下所示：

```
addLogs("Sleeper") {
    Thread.sleep(1000)
}
```

当执行处理代码时，将产生下列输出结果：

```
Sleeper started
Sleeper finished. It took 1001
```

当然，实际的输出毫秒数将稍有差异。

由于某些模式与代码库连接，因而当前模式在 Kotlin 项目中十分有用。例如，可在执行某些特性之前，检测 API 版本是否大于 Android 5.x Lollipop。对此，可使用下列条件：

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
```

```
    // Operations
}
```

但在 Kotlin 中，可通过下列方式获取函数：

```
fun ifSupportsLolipop(f:()->Unit) {
    if (Build.VERSION.SDK INT >= Build.VERSION CODES.LOLLIPOP)
    {
        f()
    }
}

// Usage
ifSupportsLollipop {
    // Operation
}
```

该方式不仅较为适宜，同时也减少了代码的冗余性，因而在具体操作中视为一种较好的操作方案。需要注意的是，该规则可定义一类控制结构，并以标准形式相类似的方式工作。例如，可定义一种简单的控制结构，若代码体中的语句未返回任何错误即可运行，其定义和应用方式如下所示：

```
fun repeatUntilError(code: ()->Unit): Throwable {
    while (true) {
        try {
            code()
        } catch (t: Throwable) {
            return t
        }
    }
}

// Usage
val tooMuchAttemptsError = repeatUntilError {
    attemptLogin()
}
```

除此之外，其优点还体现在：自定义的数据结构还可返回一个值。其中，最引人瞩目之处在于，无须其他附加语言的支持，即可定义几乎任意所需的控制结构。

5.6.2　利用 LINQ 风格处理数据结构

前述内容曾提到，Kotlin 支持 LINQ 风格的处理。参数规则中的最后一个 Lambda 表

示为另一个组件，并可提升可读性。例如，考察下列代码：

```
strings.filter { it.length == 5 }.map { it.toUpperCase() }
```

与参数规则中未使用最后一个 Lambda 相比，其可读性获得了显著的提升，如下所示：

```
strings.({ s -> s.length == 5 }).map({ s -> s.toUpperCase() })
```

再次说明，该处理过程将在第 7 章详细讨论，当前仅讨论了与改善可读性相关的两种特性（参数规则中最后一个 Lambda，以及单一参数的隐式名称）。

参数规则中最后一个 Lambda 也是 Kotlin 特征之一，经引入后可改善 Lambda 表达式的应用体验，其具体改进措施也较为丰富，相关方案的协同工作方式十分重要，旨在提升高阶函数的简洁性、可读性以及效率。

5.7　Kotlin 中的 Java SAM 支持

在 Kotlin 中，高阶函数的使用方式较为简单，其问题主要体现在与 Java 的互操作方面，而 Kotlin 对此并未提供本地支持。对此，可采用仅包含单一方法的接口予以替代，此类接口称作单一抽象方法（SAM）或函数式接口。对此，较好的示例（通过该方式构建一个函数）是在 View 元素上使用 setOnClickListener。在 Java 版本 8 之前，除了使用匿名内部类之外，尚无简单的实现方法，如下所示：

```
// Java
button.setOnClickListener(new OnClickListener() {
    @Override public void onClick(View v) {
        // Operation
    }
});
```

在上述示例中，OnClickListener 方法定义为 SAM，其中仅包含了单一方法 onClick。虽然 SAM 实际上常用作函数定义的替代方案，但 Kotlin 也对其生成了一个构造函数，并作为参数包含了函数类型，称作 SAM 构造函数。SAM 构造函数可创建 Java SAM 接口实例，即调用其名称并传递函数字面值，如下所示：

```
button.setOnClickListener(OnClickListener {
    /* ... */
})
```



函数字面值表示为一个表达式，并定义了未命名函数。在 Kotlin 中，包含两种函数字面值：

（1）匿名函数。

（2）Lambda 表达式。

两种函数字面值均可描述为：

```
val a = fun() {}        // Anonymous function
val b = {}              // Lambda expression
```

更好的方案是，针对接收 SAM 的各个 Java 方法，Kotlin 编译器将生成某个版本，并作为参数接收函数，因此可按照下列方式设置 OnClickListener：

```
button.setOnClickListener {
    // Operations
}
```

回忆一下，Kotlin 编译器仅对 Java SAM 生成 SAM 构造函数以及函数方法，且并不会针对包含单一方法的 Kotlin 接口生成 SAM 构造函数。当采用 Kotlin 编写函数并包含一个 SAM 时，则无法将其用作 Java 方法（基于参数 SAM），如下所示：

```
interface OnClick {
    fun call()
}

fun setOnClick(onClick: OnClick) {

    //...
}

setOnClick { } // 1. Error
```

在注释 1 中，由于 setOnClick 函数采用 Kotlin 编写，因而无法实现正常工作。

在 Kotlin 中，接口不应采用这一方式予以使用。对此，推荐方法是使用函数类型，而非 SAM，如下所示：

```
fun setOnClick(onClick: ()->Unit) {
    //...
}

setOnClick { } // Works
```

Kotlin 编译器针对定义于 Java 中的各个 SAM 接口生成构造函数，该接口仅包含替换某个 SAM 的函数类型。考察下列接口：

```
// Java, inside View class
public interface OnClickListener {
    void onClick(View v);
}
```

可通过下列方式在 Kotlin 中加以使用：

```
val onClick = View.OnClickListener { toast("Clicked") }
```

或者作为函数参数予以提供，如下所示：

```
fun addOnClickListener(d: View.OnClickListener) {}
addOnClickListener( View.OnClickListener { v -> println(v) })
```

下列代码显示了 Android 中 Java SAM Lambda 接口和方法示例：

```
view.setOnLongClickListener { /* ... */; true }
view.onFocusChange { view, b -> /* ... */ }

val callback = Runnable { /* ... */ }
view.postDelayed(callback, 1000)
view.removeCallbacks(callback)
```

下列代码显示了 RxJava 示例：

```
observable.doOnNext { /* ... */ }
observable.doOnEach { /* ... */ }
```

下面考察 Kotlin 中 SAM 定义替代方案的实现方式。

5.8　命名 Kotlin 函数类型

Kotlin 并不支持定义于 Java 中的 SAM 类型转换，对此，推荐方法是使用函数类型。但 SAM 对经典函数并不存在优势，即名称和命名参数。较好的方法是，若函数类型定义较长，或作为参数多次被传递，则须命名该函数类型；另外，若仅通过类型无法判断各个参数的含义，则应定义命名参数。

在后续章节中，读者将会看到，可对参数和函数类型全部定义予以命名，并通过类型别名以及函数类型中的命名参数予以实现。通过这一方式，即可在持有函数类型的前提下发挥 SAM 的各种优点。

5.8.1　函数类型中的命名参数

截止到目前，前述内容仅讨论了函数类型定义（其中，仅指定相关类型），但并未涉

及参数名。相应地，参数名已在函数字面值中加以定义，如下所示：

```
fun setOnItemClickListener(listener: (Int, View, View)->Unit) {
    // code
}
setOnItemClickListener { position, view, parent -> /* ... */ }
```

若参数不具备自解释特征，且开发人员无法了解参数的具体含义，则问题也会随之出现。当采用 SAM 时，其中包含了一些应用建议，但在前述示例中定义的函数类型中，此类建议并未起到应有的效果，如图 5.2 所示。

```
fun setOnItemClickListener(listener: (Int, View, View)->Unit) {
    // code
}
setOnItemClickListener {}
        i, view, view ->
        i: Int, view: View, view: View ->
        setOnItemClickListener(listener: (Int, View, View) -> Un…  Unit
        setOnItemClickListener { Int, View, View -> ... } (liste…  Unit
        print(message: Boolean) (kotlin.io)                       Unit
        print(message: Int) (kotlin.io)                           Unit
        print(message: Any?) (kotlin.io)                          Unit
        print(message: Byte) (kotlin.io)                          Unit
        print(message: Char) (kotlin.io)                          Unit
        print(message: Long) (kotlin.io)                          Unit
        Did you know that Quick Documentation View (Ctrl+Q) works in completion lookups as well? >>
```

图 5.2

该方案旨在利用命名参数定义函数类型，如下所示：

```
(position: Int, view: View, parent: View)->Unit
```

这一形式的优点在于，IDE 可作为函数字面值中的参数名称给予名称提示。据此，程序员消除某些歧义，如图 5.3 所示。

```
fun setOnClickListener(onClick: (position: Int, view: View, parent: View)->Unit) {
    // ...
}

fun main(args: Array<String>) {
    setOnClickListener { | }
}
        position, view, parent ->
        position: Int, view: View, parent: View ->
        main(args: Array<String>) (<root>)                        Unit
        setOnClickListener(onClick: (Int, View, View) -> Unit) (…  Unit
        setOnClickListener { position, view, parent -> ... } (on…  Unit
        tryFewTimes {...} (..., f: () -> Unit) (<root>)           Unit
        try {...}
        tryFewTimes(times: Int = ..., f: () -> Unit) (<root>)     Unit
        main(args: Array<String>) (<root>)                        Unit
        addClassic(node: BinaryTree<T>?, elem: T) (<root>)        Unit
        Did you know that Quick Definition View (Ctrl+Shift+I) works in completion lookups as well? >>
```

图 5.3

当同一函数类型多次使用时，问题便会出现，因而难以针对各个定义指定对应的参数。此时，可采用 Kotlin 中类型别名这一特性。

5.8.2　类型别名

自版本 1.1 以来，Kotlin 支持类型别名这一特性，并可针对现有类型提供替换名称。下列代码显示了类型别名定义示例，其中生成了一个 Users 列表。

```
data class User(val name: String, val surname: String)
typealias Users = List<User>
```

通过这一方式，可向现有数据类型中添加更多具有实际含义的名称，如下所示：

```
typealias Weight = Double
typealias Length = Int
```

类型别名须在顶级位置处进行声明。对此，可见修饰符可应用于类型别名，进而调整其适用范围，且默认条件下为 public 类型。这也意味着，之前定义的类型别名可无限制地加以使用，如下所示：

```
val users: Users = listOf(
    User("Marcin", "Moskala"),
    User("Igor", "Wojda")
)

fun calculatePrice(length: Length) {
    // ...
}
calculatePrice(10)

val weight: Weight = 52.0
val length: Length = 34
```

需要注意的是，别名用于改善代码的可读性；同时，仍可交互使用原始类型，如下所示：

```
typealias Length = Int
var intLength: Int = 17
val length: Length = intLength
intLength = length
```

typealias 的另一种应用则是缩短较长的泛型，并向其赋予更具含义的名称。当同一类型在代码中多处出现时，这可有效地改进代码的可读性和一致性，如下所示：

```
typealias Dictionary<V> = Map<String, V>
typealias Array2D<T> = Array<Array<T>>
```

类型别名常用于命名函数类型，如下所示：

```
typealias Action<T> = (T) -> Unit
typealias CustomHandler = (Int, String, Any) -> Unit
```

相应地，可结合函数类型参数名对其加以使用，如下所示：

```
typealias OnElementClicked = (position: Int, view: View, parent: View) ->Unit
```

随后即可查看到相应的参数提示，如图 5.4 所示。

```
typealias OnElementClicked = (position: Int, view: View, parent: View)->Unit

fun main(args: Array<String>) {
    setOnClickListener { }
}
        position, view, parent ->
        position: Int, view: View, parent: View ->
        main(args: Array<String>) (<root>)                             Unit
        setOnClickListener(onClick: OnElementClicked /* = (Int, …     Unit
        setOnClickListener { position, view, parent -> ... } (on…    Unit
        tryFewTimes {...} (..., f: () -> Unit) (<root>)               Unit
        try {...}
        tryFewTimes(times: Int = ..., f: () -> Unit) (<root>)         Unit
        main(args: Array<String>) (<root>)                             Unit
        addClassic(node: BinaryTree<T>?, elem: T) (<root>)            Unit
        add(node: BinaryTree<T>?, elem: T) (<root>)          BinaryTree<T>
Use Ctrl+Shift+Enter to syntactically correct your code after completing (balance parentheses etc.) >> π
```

图 5.4

下面考察类型别名命名的函数类型的实现方式（基于类）。在该示例中，函数类型的参数类型建议为方法参数名，如下所示：

```
typealias OnElementClicked = (position: Int, view: View, parent: View) ->Unit

class MainActivity: Activity(), OnElementClicked {

    override fun invoke(position: Int, view: View, parent: View) {
       // code
    }
}
```

使用命名函数类型的主要原因包括：

- 与全部函数类型定义相比，名字更加简短、易用。
- 当传递函数时，在修改其定义后，若使用类型别名，则无须在其他处对其进行调整。
- 使用类型别名时，可方便地设置所定义的参数名。

当结合使用两种特性后（即函数类型中的命名参数和类型别名），则无须在 Kotlin 中定义 SAM——对于函数类型而言，SAM 中的全部优点（名称和命名参数）均可通过函数

类型定义中的命名参数和参数别名予以实现。同时，这也体现了 Kotlin 对于函数式编程的另一种支持。

5.9　针对未使用变量的下划线

在某些场合下，所定义的 Lambda 表达式并不会使用其全部参数。当保留其命名状态时，可能会对程序员理解 Lambda 表达式及其功能产生负面影响。下面考察过滤第二个参数的函数。其中，第二个参数表示为元素值，在当前示例中未予使用，如下所示：

```
list.filterIndexed { index, value -> index % 2 == 0 }
```

为了消除误解，此处采用了多项约定，例如忽略参数名，如下所示：

```
list.filterIndexed { index, ignored -> index % 2 == 0 }
```

鉴于此类约定尚不完善，因而 Kotlin 引入了下划线符号，用于未使用的参数名的替代方案，如下所示：

```
list.filterIndexed { index, _ -> index % 2 == 0 }
```

此处建议使用该符号，当未使用 Lambda 表达式参数时，将显示一条警告信息，如图 5.5 所示。

```
listOf(10, 10, 10).filterIndexed { index, elem -> index % 2 == 0 }
```

Parameter 'elem' is never used, could be renamed to _

图 5.5

5.10　Lambda 表达式中的解构机制

第 4 章曾讨论了对象如何通过解构声明解构为多个属性，如下所示：

```
data class User(val name: String, val surname: String, val phone: String)

val (name, surname, phone) = user
```

自版本 1.1 起，Kotlin 针对 Lambda 参数使用了解构声明语法。对此，须采用括号包含全部需要解构的参数，如下所示：

```
val showUser: (User) -> Unit = { (name, surname, phone) ->
    println("$name $surname have phone number: $phone")
}
```

```
val user = User("Marcin", "Moskala", "+48 123 456 789")
showUser(user)
// Marcin Moskala have phone number: +48 123 456 789
```



Kotlin 的解构声明基于位置，而非基于名称的解构声明，例如 TypeScript。在基于位置的解构声明中，属性顺序决定了对应属性赋予哪个变量。在基于名称的解构机制中，则由变量名予以确定，如下所示：

```
// TypeScript
const obj = { first: 'Jane', last: 'Doe' };
const { last, first } = obj;
console.log(first); // Prints: Jane
console.log(last); // Prints: Doe
```

两种方案各具优缺点。其中，基于位置的解构声明适用于某个属性的重命名操作，但对于属性重排序而言，则缺少应有的安全性。相比较而言，基于名称的解构声明对于属性重排序来说较为安全，但不适用于属性的重命名。

解构声明可在单一 Lambda 表达式中使用多次，并可与常规参数结合使用，如下所示：

```
val f1: (Pair<Int, String>)->Unit = { (first, second) ->
    /* code */ } // 1
val f2: (Int, Pair<Int, String>)->Unit = { index, (f, s)->
    /* code */ } // 2
val f3: (Pair<Int, String>, User) ->Unit = { (f, s), (name,
    surname, tel) ->/* code */ } // 3
```

在注释 1 中，Pair 解构；在注释 2 中，Pair 和其他元素的解构；在注释 3 中，单一 Lambda 表达式中的多个解构。

注意，可将某个类解构为小于所有组件，如下所示：

```
val f: (User)->Unit = { (name, surname) -> /* code */ }
```

解构声明中支持下划线符号，常用于获取更深一层的组件，如下所示：

```
val f: (User)->Unit = { (name,  , phone) -> /* code */ }
val third: (List<Int>)->Int = { (_, _, third) -> third }
```

解构参数的类型可通过下列方式指定：

```
val f = { (name, surname): User -> /* code */ } //1
```

在注释 1 中，对应类型根据当前 Lambda 表达式进行推断；在注释 2 中，由于缺少 Lambda 表达式中与类型相关的足够信息，因而具体类型无法被推断。

这也使得 Lambda 中的解构成为一项有用的特性。下面考察 Android 中某些较为常见

的用例。该示例用于处理 Map 元素，对应元素类型为 Map.Entry，并解构为 key 和 value 参数，如下所示：

```
val map = mapOf(1 to 2, 2 to "A")
val text = map.map { (key, value) -> "$key: $value" }
println(text) // Prints: [1: 2, 2: A]
```

类似地，Pairs 列表可解构为：

```
val listOfPairs = listOf(1 to 2, 2 to "A")
val text = listOfPairs.map { (first, second) ->
    "$first and $second" }
println(text) // Prints: [1 and 2, 2 and A]
```

当尝试简化数据对象处理时，也可使用解构声明，如下所示：

```
fun setOnUserClickedListener(listener: (User)->Unit) {
    listView.setOnItemClickListener {  ,  , position,   ->
        listener(users[position])
    }
}

setOnUserClickedListener { (name, surname) ->
    toast("Clicked to $name $surname")
}
```

当用于异步处理元素时，该方案十分有用（例如 RxJava）。其中，对应函数设计为处理单一元素，若需要处理多个元素，则应将其包装至 Pair、Triple 或其他一些数据类中，同时在每个步骤上使用解构声明，如下所示：

```
getQuestionAndAnswer()
    .flatMap { (question, answer) ->
      view.showCorrectAnswerAnimationObservable(question, answer)
    }
    .subscribe( { (question, answer) -> /* code */ } )
```

5.11　内 联 函 数

高阶函数十分有效，并可改善代码的复用性。然而，其中一个最大的问题是如何高效地对其加以使用。Lambda 表达式可编译为类（通常为匿名类）；另外，Java 中对象的创建任务也是一项较为繁重的工作。对此，可通过较为高效的方式使用高阶函数，同时利用函

数内联机制保留其全部优点。

内联函数这一概念历史相对悠久，且多与 C++或 C 语言相关。当某一函数标记为内联函数时，在代码编译期间，编译器利用实际的函数体替换全部函数调用，且不再视为函数，而是真实的代码。这使得字节码较为冗长，但运行期执行将更加高效。稍后，读者将会看到，几乎源自标准库中的全部高阶函数均标记为内联函数。下列示例通过 inline 修饰符标记 printExecutionTime 函数。

```
inline fun printExecutionTime(f: () -> Unit) {
    val startTime = System.currentTimeMillis()
    f()
    val endTime = System.currentTimeMillis()
    println("It took " + (endTime - startTime))
}

fun measureOperation() {
    printExecutionTime {
        longOperation()
    }
}
```

当编译或反编译 measureOperation 时，将会发现，函数调用被真实的代码体所替代，参数函数调用则通过 Lambda 表达式的代码体被替换，如下所示：

```
fun measureOperation() {
    val startTime = System.currentTimeMillis() // 1
    longOperation() // 2
    val endTime = System.currentTimeMillis()
    println("It took " + (endTime - startTime))
}
```

在注释 1 中，源自 printExecutionTime 中的代码被添加至 measureOperation 函数体中。在注释 2 中，Lambda 内的代码位于其自身调用中。如果函数对其加以多次使用，代码将替换各次调用。



printExecutionTime 的代码体仍可在代码中看到，省略该代码旨在提升当前示例的可读性，并在编译后加以使用，因而仍保留于当前代码中。例如，若该代码作为库添加至某个项目中，而且，当通过 Kotlin 加以使用时，该函数仍将以内联方式工作。

虽然无须针对 Lambda 表达式创建类，内联函数仍可加速函数的执行过程，这一差别十分重要，因而建议针对至少包含一个参数的全部短小型函数使用内联修饰符。然而，使

用内联修饰符也包含负面影响。首先，所生成的字节码较长，这一点之前也曾有所提及。其原因在于，函数调用被函数体所替换，该函数体内的 Lambda 调用被函数字面值代码体所替换。另外，内联函数无法实现递归，且无法使用包含多个限定可见性修饰符（相比于 Lambda 表达式）的函数或类。例如，公有内联函数无法使用私有函数，其原因可描述为：可导致代码置入函数中且无法正常使用，进而产生编译错误。为了防止这一问题，Kotlin 不允许使用包含小于 Lambda 表达式的限定修饰符的元素，如下所示：

```
internal fun someFun() {}
inline fun inlineFun() {
    someFun() // ERROR
}
```

实际上，如果忽略此类警告，Kotlin 仍可在 inline 函数中使用包含多个限定可见性修饰符的元素，但这并非是良好习惯，且应禁用下列方式：

```
// Tester1.kt
fun main(args: Array<String>) { a() }

// Tester2.kt
inline fun a() { b() }
private fun b() { print("B") }
```

对于内部修饰符，其实现过程较为简单：内部修饰符表示为 public。对于私有函数，则存在一个附加的 access$b 函数，其中包含了 public 可见性修饰符，且仅调用 b 函数，如下所示：

```
public static final void access$b() { b(); }
```

该行为旨在解释：某些时候，较少的限定修饰符仍可在 inline 函数中加以使用（对应情形可在 Kotlin 1.1 标准库中看到）。在相关项目中，可利用这一方式设计元素，且无须禁用某些操作。

另一个问题则并不直观。虽然未创建 Lambda，但仍不可将具有函数类型的参数传递至另一个函数中，如下所示：

```
fun boo(f: ()->Int) {
    //...
}

inline fun foo(f: () -> Int) {
    boo (f) // ERROR, 1
}
```

若函数为 inline，那么，其函数参数无法传递至非内联函数中。

这里，具体原因可解释为：未创建 f 参数，该参数设计为由函数字面值予以替换。因此，无法作为参数传递至另一个函数中。

对此，较为简单的处理方式是：将 boo 函数也定义为内联函数。在大多数时候，内联函数的数量不可过多，其原因包括：

- inline 函数应用于小型函数。如果定义了使用其他 inline 函数的 inline 函数，编译后将生成较大的结构，这一问题源自编译器以及最终的代码尺寸。
- 对于包含超量可见性修饰符的元素，inline 函数无法对其加以使用；若在库中使用（多个函数为私有函数且对 API 予以保护），这仍将会产生问题。

对于此类问题，最为简单的处理方式是：定义相关参数，且传递至另一个 noinline 函数中。

5.11.1 noinline 修饰符

noinline 是针对函数类型参数定义的修饰符，并将特定参数视为常规函数类型参数（其调用并不会被函数字面值替换）。noinline 函数示例如下所示：

```
fun boo(f: ()->Unit) {
    //...
}

inline fun foo(before: ()->Unit, noinline f: () -> Unit) { // 1
    before() // 2
    boo (f) // 3
}
```

在注释 1 中，noinline 注释修饰符位于参数 f 之前；在注释 2 中，before 函数被 Lambda 表达式体（用作参数）替换；在注释 3 中，f 表示为 noinline，因而可被传递至 boo 函数中。

使用 noinline 的主要原因在于：

- 需要向其他函数传递特定的 Lambda。
- 频繁调用 Lambda，但不希望增加代码量。

需要注意的是，若将全部函数参数均标记为 noinline，内联函数的性能基本毫无变化。虽然使用 inline 并不会获得收益，但编译器还是会显示一条警告消息。因此，noinline 大多数时候仅用于多个函数参数，同时仅将其用于部分参数。

5.11.2 非本地返回

包含函数参数的函数其行为类似于本地结构（例如循环结构）。前述内容已经查看了

ifSupportsLolipop 和 repeatUntilError 函数。更为常见的示例则是 forEach 修饰符，并可作为 for 控制结构的替代方案，同时依次调用包含各元素的参数函数。下列代码展示了实现过程（Kotlin 标准库中存在 forEach 修饰符，但稍后将会看到，其中包含了尚未出现的元素）：

```
fun forEach(list: List<Int>, body: (Int) -> Unit) {
    for (i in list) body(i)
}
// Usage
val list = listOf(1, 2, 3, 4, 5)
forEach(list) { print(it) } // Prints: 12345
```

这里的问题是，通过当前方式定义的 forEach 函数，将无法从外部函数中返回。例如，下列代码显示了基于 for 循环的 maxBounded 函数实现方式：

```
fun maxBounded(list: List<Int>, upperBound: Int, lowerBound: Int):
Int {
    var currentMax = lowerBound
    for(i in list) {
        when {
            i > upperBound -> return upperBound
            i > currentMax -> currentMax = i
        }
    }
    return currentMax
}
```

如果希望将 forEach 视为 for 循环的替代方案，类似的情形还将会出现并被支持。具体问题将体现为：代码无法被编译（使用 forEach 而非 for 循环），如图 5.6 所示。

```
fun maxBounded(list: List<Int>, upperBound: Int, lowerBound: Int): Int {
    var currentMax = lowerBound
    forEach(list) { i ->
        when {
            i > upperBound -> return upperBound
      [RETURN_NOT_ALLOWED] 'return' is not allowed here
    }
    return currentMax
}
```

图 5.6

具体原因与代码的编译方式有关。如前所述，Lambda 表达式编译为匿名对象类，其中包含了代码定义的方法。由于处于不同的上下文环境中，因而无法从 maxBounded 函数中返回。

当遇到 forEach 函数标记为 inline 这种情况时，之前也曾谈到，函数体将在编译时替换

其调用，参数中的全部函数将利用对应的代码体予以替换。因此，此处使用 return 修饰符并无问题。随后，若 forEach 为 inline，则可在 Lambda 表达式内部使用 return，如下所示：

```
inline fun forEach(list: List<Int>, body: (Int)->Unit) {
    for(i in list) body(i)
}

fun maxBounded(list: List<Int>, upperBound: Int,
    lowerBound: Int): Int {
    var currentMax = lowerBound
    forEach(list) { i ->
        when {
            i > upperBound -> return upperBound
            i > currentMax -> currentMax = i
        }
    }
    return currentMax
}
```

这也体现了 Kotlin 中 maxBounded 函数的编译方式；当反编译至 Java 后（经过适当简化），对应代码如下所示：

```
public static final int maxBounded(@NotNull List list,
int upperBound, int lowerBound) {
    int currentMax = lowerBound;
    Iterator iter = list.iterator();
    while(iter.hasNext()) {
        int i = ((Number)iter.next()).intValue();
        if(i > upperBound) {
            return upperBound; // 1
        }

        if(i > currentMax) {
            currentMax = i;
        }
    }

    return currentMax;
}
```

在上述代码中，return 十分重要——该语句定义于 Lambda 表达式中，并从 maxBounded 函数中返回。

inline 函数的 Lambda 表达式中的 return 修饰符称作非本地返回。

5.11.3 Lambda 表达式中的标记返回

下面考察 Lambda 表达式（而非函数）中的返回。对此，可使用标记实现这一任务。基于标记的 Lambda 表达式返回如下所示：

```
inline fun <T> forEach(list: List<T>, body: (T) -> Unit) { // 1
    for (i in list) body(i)
}

fun printMessageButNotError(messages: List<String>) {
    forEach(messages) messageProcessor@ { // 2
        if (it == "ERROR") return@messageProcessor // 3
        print(it)
    }
}

// Usage
val list = listOf("A", "ERROR", "B", "ERROR", "C")
processMessageButNotError(list) // Prints: ABC
```

注释 1 可视为 forEach 函数的通用实现，其中可处理包含任意类型的列表。在注释 2 中，则在 forEach 参数中针对 Lambda 表达式定义标记。注释 3 则从标记指定的 Lambda 表达式中返回。

Kotlin 的另一个特性是，定义为函数参数的 Lambda 表达式包含默认标记，其名称等同于其中所定义的函数。相应地，该标记称作隐式标记。当需要从定义在 forEach 函数中的 Lambda 表达式中返回时，可使用 return@forEach 完成这一操作。对应示例如下所示：

```
inline fun <T> forEach(list: List<T>, body: (T) -> Unit) { // 1
    for (i in list) body(i)
}

fun processMessageButNotError(messages: List<String>) {
    forEach(messages) {
        if (it == "ERROR") return@forEach // 1
        process(it)
    }
}

// Usage
val list = listOf("A", "ERROR", "B", "ERROR", "C")
processMessageButNotError(list) // Prints: ABC
```

在注释 1 中，隐式标记名从函数名中获取。

注意，虽然 forEach 为 inline，但还可使用非本地返回，进而从 processMessageButNotError 函数中返回，如下所示：

```
inline fun <T> forEach(list: List<T>, body: (T) -> Unit) {
    for (i in list) body(i)
}

fun processMessageButNotError(messages: List<String>) {
    forEach(messages) {
        if (it == "ERROR") return
        process(it)
    }
}

// Usage
val list = listOf("A", "ERROR", "B", "ERROR", "C")
processMessageButNotError(list) // Prints: A
```

下面考察使用本地返回标记的更为复杂的示例。假设包含两个 forEach 循环，且一个循环位于另一个循环中。当使用隐式标记时，将从更深的循环中返回。在当前示例中，可采用这一方式，并忽略特定消息的处理过程，如下所示：

```
inline fun <T> forEach(list: List<T>, body: (T) -> Unit) { // 1
    for (i in list) body(i)
}

fun processMessageButNotError(conversations: List<List<String>>) {
    forEach(conversations) { messages ->
        forEach(messages) {
            if (it == "ERROR") return@forEach // 1.
            process(it)
        }
    }
}

// Usage
val conversations = listOf(
    listOf("A", "ERROR", "B"),
    listOf("ERROR", "C"),
    listOf("D")
)
processMessageButNotError(conversations) // ABCD
```

在注释 1 中，将从定义在 forEach 函数（该函数接收消息作为参数）中的 Lambda 返回。

在使用隐式标记的同一上下文中，无法从另一个 Lambda 表达式中返回——该标记位于更深的隐式标记下。

在此类情形中，需要使用非本地隐式标记返回，且仅支持内联函数参数。在当前示例中，虽然 foEach 为内联函数，但可通过下列方式从函数字面值中返回：

```
inline fun <T> forEach(list: List<T>, body: (T) -> Unit) { // 1
    for (i in list) body(i)
}

fun processMessageButNotError(conversations: List<List<String>>) {
    forEach(conversations) conv@ { messages ->
        forEach(messages) {
            if (it == "ERROR") return@conv // 1.
            print(it)
        }
    }
}

// Usage
val conversations = listOf(
    listOf("A", "ERROR", "B"),
    listOf("ERROR", "C"),
    listOf("D")
)
processMessageButNotError(conversations) // AD
```

针对注释 1，将从定义于 forEach 中的 Lambda 返回。

除此之外，还可使用非本地返回（不包含任何标记的返回）结束当前处理过程，如下所示：

```
inline fun <T> forEach(list: List<T>, body: (T) -> Unit) { // 1
    for (i in list) body(i)
}

fun processMessageButNotError(conversations: List<List<String>>) {
    forEach(conversations) { messages ->
        forEach(messages) {
            if (it == "ERROR") return // 1.
            process(it)
        }
    }
}
```

对于注释 1，这将从 processMessageButNotError 函数中返回，并结束当前处理过程。

5.11.4　crossinline 修饰符

某些时候，需要从内联函数（未直接位于函数体中）中使用函数类型参数，但位于另一个执行环境中，例如本地对象或嵌套函数。但是，由于支持非本地返回，内联函数的标准函数类型参数不允许通过这种方式加以使用；如果该函数可在另一个执行环境下使用，则应予以禁用。为了通知编译器非本地返回禁用，对应参数须标记为 crossinline。随后，即可在 inline 函数中实现替代作用，甚至可在另一个 Lambda 表达式中使用，如下所示：

```
fun boo(f: () -> Unit) {
    //...
}

inline fun foo(crossinline f: () -> Unit) {
    boo { println("A"); f() }
}

fun main(args: Array<String>) {
    foo { println("B") }
}
```

这将编译为下列内容：

```
fun main(args: Array<String>) {
    boo { println("A"); println("B") }
}
```

虽然利用该函数并未生成属性，但不可将 crossinline 参数作为参数传递至另一个函数中，如图 5.7 所示。

```
inline fun foo(crossinline f: () -> Unit) {
    boo(f)
Illegal usage of inline-parameter 'f' in 'public inline fun foo(crossinline f: () -> Unit): Unit defined in demo'. Add 'noinline' modifier to the parameter declaration
```

图 5.7

下面考察 Android 中的实际用例。由于可使用 Looper 类中的 getMainLooper 静态函数获得主循环，因而无须 Context 执行主应用程序线程上的操作。因此，可编写顶级函数，并将单线程调整为主线程。为了对此进行优化，首先检查当前线程是否为主线程。若是，则调用相关操作；否则，生成一个处理程序，在主线程和后处理中执行操作，并调用该处理程序。为了提高该函数的执行速度，应将 runOnUiThread 函数定义为 inline，并于随后从另一个线程中执行操作调用，对此需要将其设置为 crossinline。函数的实现过程如下所示：

```
inline fun runOnUiThread(crossinline action: () -> Unit) {
    val mainLooper = Looper.getMainLooper()
    if (Looper.myLooper() == mainLooper) {
        action()
    } else {
        Handler(mainLooper).post { action() } // 1
    }
}
```

针对注释 1，鉴于 crossinline，因而可在 Lambda 表达式中运行 action。

由于可在 Lambda 表达式或本地函数上下文中使用函数类型，同时保留 inline 函数的优点（在该上下文环境下，无须创建 Lambda），因而 crossinline 十分有用。

5.11.5　inline 属性

自 Kotlin 1.1 起，inline 修饰符可用于相关属性上（无须包含幕后字段）。inline 修饰符可应用于单一访问器上，以使代码体被替换；或者还可用于整体属性上，其结果等同于将两个访问器定义为 inline。下面设置 inline 属性，用于检测和修改元素的可见性。其中，两个访问器均为 inline，其实现过程如下所示：

```
var viewIsVisible: Boolean
inline get() = findViewById(R.id.view).visibility == View.VISIBLE
inline set(value) {
  findViewById(R.id.view).visibility = if (value) View.VISIBLE
  else View.GONE
}
```

如果将整体属性标记为 inline，则可获得相同结果，如下所示：

```
inline var viewIsVisible: Boolean
get() = findViewById(R.id.view).visibility == View.VISIBLE
  set(value) {
    indViewById(R.id.view).visibility = if (value) View.VISIBLE
      else View.GONE
    }

// Usage
if (!viewIsVisible)
viewIsVisible = true
```

上述代码可编译为下列结果：

```
if (!(findViewById(R.id.view).getVisibility() == View.VISIBLE))
{
```

```
  findViewById(R.id.view).setVisibility(true?View.VISIBLE:View.GONE);
}
```

通过这一方式，可忽略 set 和 get 方法调用，并在增加编译代码尺寸的前提下提升操作性能。对于大多数属性，使用 inline 修饰符依然可获得较大收益。

5.12　函 数 引 用

某些时候，需要作为参数传递的函数已经定义为独立的函数。随后，则可通过其调用定义 Lambda，如下所示：

```
fun isOdd(i: Int) = i % 2 == 1

list.filter { isOdd(it) }
```

在 Kotlin 中，函数可作为值进行传递。为了将顶级函数用作某个值，需要使用到函数引用，即使用两个冒号以及函数名（形如::functionName）。下列代码显示了其应用方式，进而提供断言以执行过滤操作：

```
list.filter(::isOdd)
```

对应示例如下所示：

```
fun greet(){
    print("Hello! ")
}

fun salute(){
    print("Have a nice day ")
}

val todoList: List<() -> Unit> = listOf(::greet, ::salute)

for (task in todoList) {
    task()
}

// Prints: Hello! Have a nice day
```

函数引用可视作反射机制的一个示例，因此，该操作返回的对象同样包含了与引用函数相关的信息，如下所示：

```
fun isOdd(i: Int) = I % 2 == 1
```

```
val annotations = ::isOdd.annotations
val parameters = ::isOdd.parameters
println(annotations.size) // Prints: 0
println(parameters.size) // Prints: 1
```

此处，该对象还实现了函数类型，并可通过下列方式加以使用：

```
val predicate: (Int)->Boolean = ::isOdd
```

另外，还可对方法加以引用。对此，须编写类型名、两个冒号以及方法名（形如 Type::functionName），对应示例如下所示：

```
val isStringEmpty: (String)->Boolean = String::isEmpty

// Usage
val nonEmpty = listOf("A", "", "B", "")
.filter(String::isNotEmpty)
print(nonEmpty) // Prints: ["A", "B"]
```

在上述示例中，当引用非静态方法时，须提供类实例作为参数。相应地，isEmpty 函数定义为 String 方法，且未接收任何参数。isEmpty 引用则包含一个 String 参数，并用作函数调用上的一个对象。该对象的引用通常位于第一个参数。下列代码显示了另一个示例，其中，对应方法包含了已经定义的 food 属性：

```
class User {

    fun wantToEat(food: Food): Boolean {
        // ...
    }
}

val func: (User, Food) -> Boolean = User::wantToEat
```

当引用 Java 中的静态方法时，情况则有所不同，其原因在于：此处并不需要使用到类实例。这类似于对象方法，或者伴生对象。其中，对象事先已知且无须提供该对象。此时，存在一个与引用函数相同的参数，以及相同返回类型生成的函数，如下所示：

```
object MathHelpers {
    fun isEven(i: Int) = i % 2 == 0
}

class Math {
    companion object {
```

```
        fun isOdd(i: Int) = i % 2 == 1
    }
}

// Usage
val evenPredicate: (Int)->Boolean = MathHelpers::isEven
val oddPredicate: (Int)->Boolean = Math.Companion::isOdd

val numbers = 1..10
val even = numbers.filter(evenPredicate)
val odd = numbers.filter(oddPredicate)
println(even) // Prints: [2, 4, 6, 8, 10]
println(odd) // Prints: [1, 3, 5, 7, 9]
```

在使用函数引用时，存在一类常见用例，其中，需要使用函数引用，并从所引用的某个类中提供相关方法。例如，当需要获取某些操作作为同一类中的方法时，或者需要从引用成员函数（源自所引用的类）引用函数时。另外一个简单的示例是，网络操作后所执行的任务。对应内容定义在某个 Presenter 中（例如 MainPresenter），但引用了全部 View 操作，并通过 view 属性加以定义（例如 MainView 类型），如下所示：

```
getUsers().smartSubscribe (
    onStart = { view.showProgress() }, // 1
    onNext = { user -> onUsersLoaded(user) }, // 2
    onError = { view.displayError(it) }, // 1
    onFinish = { view.hideProgress() } // 1
)
```

针对注释 1，showProgress、displayError 以及 hideProgress 均定义于 MainView 中；针对注释 2，onUsersLoaded 表示为定义于 MainPresenter 中的方法。

针对于此，Kotlin 自版本 1.1 后引入了绑定引用（bound references），并提供了与特定类绑定的引用。据此，该对象无须通过某个参数予以提供。当使用这一标记时，可通过下列方式替换之前的定义，如下所示：

```
getUsers().smartSubscribe (
    onStart = view::showProgress,
    onNext = this::onUsersLoaded,
    onError = view::displayError,
    onFinish = view::hideProgress
)
```

另一个需要引用的函数则是构造函数。例如，从数据传输对象（DTO）映射至模型中的某个类，如下所示：

```
fun toUsers(usersDto: List<UserDto>) = usersDto.map { User(it) }
```

此处，用户需要包含一个构造函数，并定义了基于 UserDto 的构造方式。



DTO 表示为一个对象，并加载进程间的数据，其用途主要是因为：某个系统（位于 API 中）间通信过程中所使用的类一般不同于当前系统（模型）内部所用的实际类。

在 Kotlin 中，构造函数视为与函数类似。另外，还可采用双冒号和类名对其加以引用，如下所示：

```
val mapper: (UserDto)->User = ::User
```

通过这种方式，可利用包含构造函数引用的某个构造函数调用替换 Lambda，如下所示：

```
fun toUsers(usersDto: List<UserDto>) = usersDto.map(::User)
```

使用函数引用而非 Lambda 表达式可生成简短且兼具可读性的标记。当作为参数传递多个函数时，该方法尤为有效。在其他情况下，绑定引用也十分有用，并可提供与特定对象绑定的引用。

5.13 本章小结

本章讨论了作为“第一公民”的函数，函数字面值（匿名类和 Lambda 表达式）的定义方式，以及在函数引用这一前提下，任意函数均可用作对象这一概念。另外，本章还介绍了高阶函数，以及 Kotlin 对此予以支持的各种特性，包括单一参数的隐式名称、参数规则中的最后一个 Lambda、Java SAM 支持、未用变量的下划线使用，以及 Lambda 表达式中的解构声明。此类特性针对高阶函数提供了极大的支持，并进一步丰富了函数的内容。

第 6 章将讨论 Kotlin 中的通用型任务，其中涉及功能强大的类和函数。除此之外，还将探讨与高阶函数结合使用时，相关类和函数的应用方式。

第 6 章 泛　　型

第 5 章讨论了函数编程，以及作为“一等公民”的函数方面的概念。

本章将介绍泛型和泛型函数等概念，并考察其存在的原因和应用方式，包括定义泛型类、接口和函数。另外，本章还将探讨运行期内泛型的处理方式、子类型关系以及泛型可空类型。

本章主要涉及以下内容：

- 泛型类。
- 泛型接口。
- 泛型函数。
- 泛型约束条件。
- 泛型可空类型。
- 变型。
- 使用点变型和声明点变型。
- 声明点目标。
- 类型擦除（type erasure）。
- 具体化和擦除类型参数。
- 星号投射语法。

6.1　泛型概述

泛型是一种编程风格，其中，类、函数、数据结构或者算法的编写方式采用了如下方案：实际类型于稍后指定。总体而言，泛型提供了类型安全特性，以及针对各种数据类型的、特定代码结构的复用行为。

Java 和 Kotlin 均对泛型提供了支持，其工作方式也较为类似。但相对于 Java，Kotlin 提供了一些改进措施，例如使用位置变异、星号投射语法，以及具体化类型参数。本章将对此加以讨论。

程序员常常需要一种方式指定包含某种特定类型的集合，例如 Int、Student 或 Car。如果未使用泛型，则需要针对各个数据类型（IntList、StudentList 以及 CarList 等）单独定义相关类，这些类包含类似的内部实现，仅存储的数据类型有所不同。这意味着，我们需要多次编写相同的代码（例如从集合中添加或删除数据项），并单独维护各个类。鉴于工作

量较大，因此，在泛型出现以前，程序员一般需要对通用列表进行操作，并在每次访问时对数据进行类型转换，如下所示：

```
// Java
ArrayList list = new ArrayList();
list.add(1);
list.add(2);
int first = (int) list.get(0);
int second = (int) list.get(1);
```

类型转换可视作是一类样板操作，当某个数据元素添加至集合时，并不存在相应的类型验证机制。针对于此，泛型给出了解决方案——泛型类定义并使用了占位符而非实际类型，该占位符称作类型参数。下面定义第一个泛型类，如下所示：

```
class SimpleList<T> // T is type parameter
```

类型参数表示当前类将使用特定类型，但对应类型将在类构建时指定。通过这种方式，可针对各种类型实例化 SimpleList 类。利用类型参数，可对包含各种数据类型的泛型类实现参数化操作，并可从单一类中创建多种数据类型，如下所示：

```
 // Usage
var intList: SimpleList<Int>
var studentList: SimpleList<Student>
var carList:SimpleList<Car>
```

SimpleList 类利用类型参数（Int、Student 以及 Car）进行参数化，进而定义了存储于既定列表中的数据类型。

函数包含了形式参数（函数声明中的变量）以及实际参数（传递至函数中的实际参数），类似技术也可以应用于泛型中。针对泛型中声明的类型，类型形参表示为一个蓝本或占位符；而类型实参则表示用于参数化某个泛型的实际类型。

在方法签名中可使用类型形参，据此，可确保向列表中添加特定类型的数据项，并从特定类型中获取数据项，如下所示：

```
class SimpleList<T> {

   fun add(item:T) { // 1
       // code
}
   fun get(intex: Int): T { // 2
       // code
   }
}
```

针对注释 1，泛型参数 T 用作数据项类型；在注释 2 中，类型参数用作返回类型。

根据类型实参，可向列表中添加数据项，或者从列表中获取数据项，相关示例如下所示：

```
class Student(val name: String)
val studentList = SimpleList<Student>()
studentList.add(Student("Ted"))
println(studentList.getItemAt(0).name)
```

从当前列表中，可添加或获取类型为 Student 的数据项。编译器将自动执行所有必需的类型检测，确保集合仅包含特定类型的对象。另外，向 add 方法中传递不兼容类型的对象将产生编译期错误，如下所示：

```
var studentList: SimpleList<Student>
studentList.add(Student("Ted"))
studentList.add(true) // error
```

鉴于期望类型为 Student，因而无法加入 Boolean 类型。



Kotlin 标准库在 kotlin.collections 包中定义了各种泛型集合，例如 List、Set 以及 Map，第 7 章将对此深入讨论。

在 Kotlin 中，泛型常与高阶函数（参见第 5 章）以及扩展函数（参见第 7 章）结合使用，相关示例包括 map、filter 以及 takeUntil 等函数，并可执行包含细节差异的各种常见操作。例如，利用 filter 函数在集合中查找匹配元素，以及指定匹配元素间的检测方式，如下所示：

```
val fruits = listOf("Babana", "Orange", "Apple", "Blueberry")
val bFruits = fruits.filter { it.startsWith("B") } //1
println(bFruits) // Prints: [Babana, Blueberry]
```

针对注释 1，可调用 startsWith 方法，其原因在于，集合仅包含 Strings，因而 Lambda 参数（it）包含相同类型。

6.2 泛型约束条件

默认条件下，可利用任何类型参数对泛型类进行参数化，但还是需要对其予以适当限制。当限定类型参数的可能值时，需要定义类型参数约束（bound）。其中，较为常见的约束类型则是上限约束。默认状态下，全部类型参数均包含 Any?作为隐式上限。因此，下列两项声明彼此等价：

```
class SimpleList<T>
class SimpleList<T: Any?>
```

上述约束表明，针对 SimpleList 类（包括可空类型），可采用任意所需类型作为类型参数——全部可空或非空类型均为 Any?的子类型，如下所示：

```
class SimpleList<T>
class Student
// usage

var intList = SimpleList<Int>()
var studentList = SimpleList<Student>()
var carList = SimpleList<Boolean>()
```

在某些场合下，须限制用作类型参数的数据类型。对此，须显式定义类型参数上限约束。针对 SimpleList 类，此处假设仅须使用数字类型作为类型参数，如下所示：

```
class SimpleList<T: Number>
// usage

var numberList = SimpleList<Number>()
var intList = SimpleList<Int>()
var doubleList = SimpleList<Double>()
var stringList = SimpleList<String>() //error
```

其中，Number 表示为抽象类，即 Kotlin 数字类型（Byte、Short、Int、Long、Float 以及 Double）的超类，并可将 Number 类及其全部子类（Int、Double 等）用作类型参数，但却无法使用 String 类——该类并非是 Number 的子类。IDE 以及编译器将视这一类不兼容类型为错误。另外，类型参数还可与 Kotlin 可空类型进行整合。

当定义包含非约束类型参数的某个类时，可采用非空和可空类型作为类型参数。少数时候，还须确保特定泛型不被可空类型参数进行参数化。为了禁用可空类型作为类型参数，需要显式地定义非空类型参数上限约束，如下所示：

```
class Action (val name:String)
class ActionGroup<T : Action>

// non-nullable type parameter upper bound

var actionGroupA: ActionGroup<Action>
var actionGroupB: ActionGroup<Action?> // Error
```

其中，不可向 ActionGroup 类传递可空类型参数（Action?）。

下面考察另一个示例。假设需要获取 ActionGroup 中最后一个 Action，下列代码简单地定义了 last 方法。

```
class ActionGroup<T : Action>(private val list: List<T>) {
    fun last(): T = list.last()
}
```

当向构造函数传递一个空表时，具体情况如下所示：

```
val actionGroup = ActionGroup<Action>(listOf())

//...
val action = actionGroup.last
// error: NoSuchElementException: List is empty

println(action.name)
```

由于不存在包含该列表索引的元素，last 方法将抛出一个错误。相比于异常，当列表为空时，可优先选用 null 值。Kotlin 标准库中已定义了一个对应的方法，并返回 null 值，如下所示：

```
class ActionGroup<T : Action>(private val list: List<T>) {
    fun lastOrNull(): T = list.lastOrNull() //error
}
```

鉴于最后一个方法将返回 null（列表中不存在可返回的元素），因而上述代码无法被编译。为了解决此类问题，须强制执行可空返回类型，即向类型参数使用位置添加一个问号（T?），如下所示：

```
class ActionGroup<T : Action>(private val list: List<T>) { // 1
    fun lastOrNull(): T? = list.lastOrNull() // 2
}
```

对于注释 1，表示为类型参数使用位置（将代码置于类型参数声明处）；对于注释 2，表示为类型参数使用位置（将代码置于类型参数使用处）。

T?参数表示，lastOrNull 应一直保持可空，且无须考虑潜在的类型参数可空性。需要注意的是，由于需要存储非空类型，并仅对特定场合（例如最后一个元素不存在）处理可空性，因而可将类型参数 T 约束恢复为非空类型 Action。随后可使用更新后的 ActionGroup 类，如下所示：

```
val actionGroup= ActionGroup<Action>(listOf())
val actionGroup = actionGroup.lastOrNull()
```

```
// Inferred type is Action?
println(actionGroup?.name) // Prints: null
```

注意，actionGroup 推断类型为可空型，即使利用非空类型参数实现了泛型的参数化。

使用位置处的可空类型并不妨碍在声明位置使用非空类型，如下所示：

```
open class Action
class ActionGroup<T : Action?>(private val list: List<T>) {
    fun lastOrNull(): T? = list.lastOrNull()
}

// Usage
val actionGroup = ActionGroup(listOf(Action(), null))
println(actionGroup.lastOrNull()) // Prints: null
```

综上所述，可针对类型参数指定非空约束，以禁止 ActionGroup 类的参数化操作（利用可空类型作为类型参数）。同时，可通过非空类型参数 Action 对 ActionGroup 类进行参数化。最后，若列表中不存在元素，最后一个属性可返回 null，因而可在使用位置处强制执行类型参数的可空性（T?）。

6.3 变　型

子类型是 OOP 编程中的一个常见概念，通过扩展类，可在两个类之间定义继承机制，如下所示：

```
open class Animal(val name: String)
class Dog(name: String): Animal(name)
```

类 Dog 扩展了 Animal 类，因而类型 Dog 表示为 Animal 的子类型。这表明，当需要使用到类型 Animal 的表达式时，可使用 Dog 类型。例如，可将其作为函数参数，或者将 Dog 类型变量赋予 Animal 类型变量，如下所示：

```
fun present(animal: Animal) {
    println( "This is ${ animal. name } " )
}
present(Dog( "Pluto" )) // Prints: This is Pluto
```

在介绍后续内容之前，首先需要讨论类和类型之间的差异。这里，类型是一类更加通用的术语，可通过类或接口定义，或者为语言的内置内容（基础类型）。在 Kotlin 中，对于各个类（例如 Dog），至少包含两种可能的类型，即非空（DOG）和可空类型（Dog？）。而且，对于各个泛型类（例如 class Box<T>），还可定义多个数据类型（Box<Dog>，

Box<Dog?>，Box<Animal>，Box<Box<Dog>>等）。

上述示例仅使用了简单类型。变型（型变）定义了复杂类型间子类型（例如 Box<Dog>和 Box<Animal>）与组件间子类型（例如 Animal 和 Dog）之间的关系方式。

在 Kotlin 中，默认条件下，泛型处于不可变状态，这也意味着，泛型 Box<Dog>和 Box<Animal>之间不存在子类型关系。Dog 组件表示为 Animal 的子类型，但 Box<Dog>并不是一个子类型，同时也并非是 Box<Animal>的子类型，如下所示：

```
class Box<T>
open class Animal
class Dog : Animal()

var animalBox = Box<Animal>()
var dogBox = Box<Dog>()

// one of the lines below line must be commented out,
// otherwise Android Studio will show only one error
animalBox = dogBox // 2, error
dogBox = animalBox // 1, error
```

对于注释 1，错误类型不匹配。需要 Box<Animal>，实际为 Box<Dog>。对于注释 2，错误类型不匹配，需要 Box<Dog>，实际为 Box<Animal>。

Box<Dog>类型即非子类型，也非 Box<Animal>的子类型，因此无法使用代码中的赋值操作。

对此，可定义 Box<Dog>和 Box<Animal>之间的子类型关系。在 Kotlin 中，泛型的子类型关系可表示为协变、逆变或者不可变状态。

若子类型关系为协变，则表明子类型被保留。泛型将持有与类型参数相同的关系。若 Dog 为 Animal 的子类型，那么 Box<Dog>则表示为 Box<Animal>的子类型。

逆变则与协变相反，并逆转子类型。泛型相对于类型参数具有反向关系。若 Dog 表示为 Animal 的子类型，则 Box<Animal>表示为 Box<Dog>的子类型。图 6.1 显示了全部变型关系。

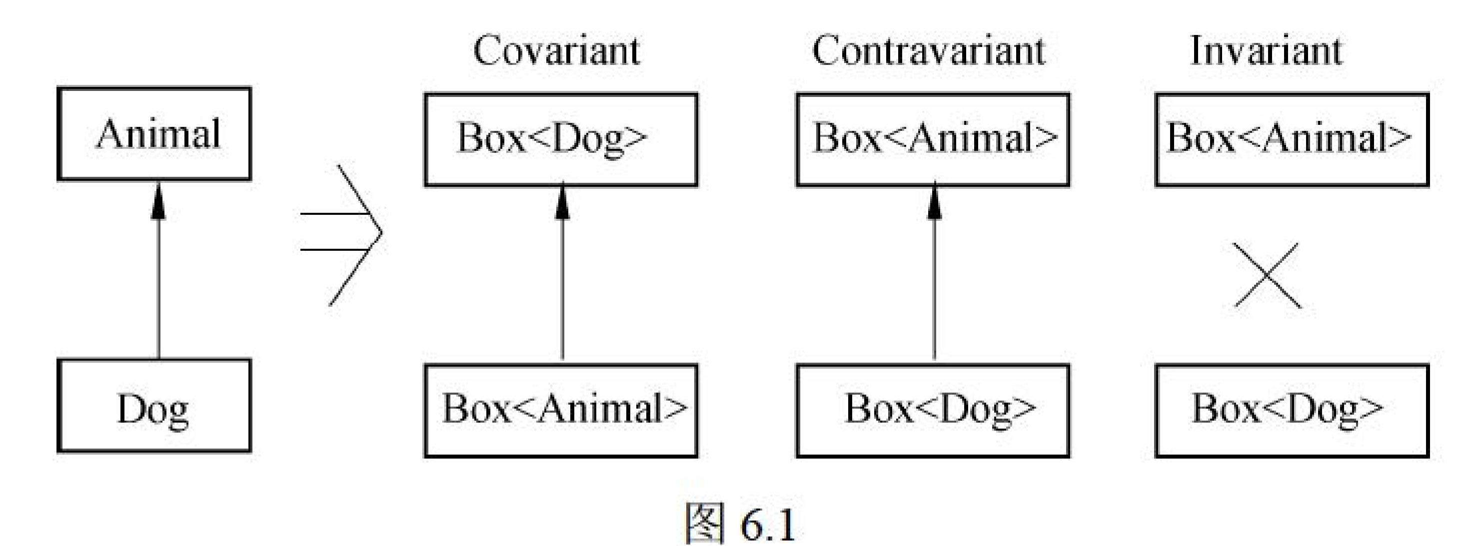


图 6.1

当定义协变或逆变行为时，需要使用到变型修饰符。

6.3.1　变型修饰符

默认条件下，Kotlin 中的变型处于不可变状态，这也表明，须将类型用作声明变量或函数参数的类型，如下所示：

```
public class Box<T> { }
fun sum(list: Box<Number>) { /* ... */ }

// Usage
sum(Box<Any>()) // Error
sum(Box<Number>()) // Ok
sum(Box<Int>()) // Error
```

此处不可使用基于 Int 的参数化泛型（表示为 Number 的子类型），也不可使用基于 Any 的参数化泛型（表示为 Number 的超类型）。通过变型修饰符，可适当放松限制，并调整某人变型。在 Java 中，问号标记（通配符符号）用于表示未知类型。据此，可定义两种通配符约束类型，即上界约束和下界约束。在 Kotlin 中，可通过 in 和 out 修饰符实现类似的行为。

在 Java 中，上约束通配符可定义接收任意参数（其子类型的特定类型）的函数。在下列示例中，sum 函数将接收任意 List，该 List 利用 Number 类或 Number 类的子类型（Box<Integer>，Box<Double>等）进行参数化，如下所示：

```
// Java
public void sum(Box<? extends Number> list) { /* ... */ }

// Usage
sum(new Box<Any>()) // Error
sum(new Box<Number>()) // Ok
sum(new Box<Int>()) // Ok
```

此处，可将 Box<Number>传递至 sum 函数以及全部子类型中，例如 Box<Int>。这一 Java 操作行为对应于 Kotlin 的 out 修饰符，并体现了协变行为，同时将当前类型限定为特定类型或当前类型的子类型。这意味着，可安全地传递 Box 类实例（利用 Number 的直接或间接子类），如下所示：

```
class Box<T>
fun sum(list: Box<out Number>) { /* ... */ }

// usage
sum(Box<Any>()) // Error
```

```
sum(Box<Number>()) // Ok
sum(Box<Int>()) // Ok
```

在 Java 中，下界约束通配符可定义接收任意参数（表示为特定类型或其子类型）的函数。在下面的示例中，sum 函数接收任意 List，该 List 利用 Number 类或 Number 类的超类型（Box<Number>和 Box<Object>）进行参数化，如下所示：

```
// Java
public void sum(Box<? super Number> list) { /* ... */ }

// usage
sum(new Box<Any>()) // Ok
sum(new Box<Number>()) // Ok
sum(new Box<Int>()) // Error
```

当前，可向 sum 函数以及全部子类型传递 Box<Any>，例如 Box<Any>。这一 Java 操作行为对应于 Kotlin 中的 in 操作符，并体现了逆变状态，同时将当前类型限定为特定类型或当前类型的超类型，如下所示：

```
class Box<T>
fun sum(list: Box<in Number>) { /* ... */ }

// usage
sum(Box<Any>()) // Ok
sum(Box<Number>()) // Ok
sum(Box<Int>()) // Error
```

此处禁止使用 in 和 out 修饰符。另外，可通过两种不同方式定义变型修饰符，稍后将对此加以讨论。

6.3.2　使用位置变型和声明位置变型

使用位置和声明位置变型基本上描述了代码的位置（地点），其中指定了变型修饰符。下面考察 View 和 Presenter 示例，如下所示：

```
interface BaseView
interface ProductView : BaseView
class Presenter<T>

// Usage
var preseter = Presenter<BaseView>()
var productPresenter = Presenter<ProductView>()
preseter = productPresenter
```

```
// Error: Type mismatch
// Required: Presenter<BaseView>
// Found: Presenter<ProductView>
```

Presenter 在其类型 parameterT 处于不可变状态。为了修正这一问题，可显式地定义子类型关系。其中包括两种方式（使用点和声明点）。下面首先定义使用点处的变型，如下所示：

```
var preseter: Presenter<out BaseView> = Presenter<BaseView>() //1
var productPresenter = Presenter<ProductView>()
preseter = productPresenter
```

针对注释 1，变型修饰符在类型参数使用点处定义了变型修饰符。

当前，preseter 变量可存储 Presenter<BaseView>的子类型，包括 Presenter<ProductView>。当前方案可正常工作，但需要对其实现加以改进。该方案包含两个问题。目前，在每次使用泛型时，均需要指定 out 变型修饰符。例如，在不同类以及多个变量中对其加以使用，如下所示：

```
// Variable declared inside class A and class B

var preseter = Presenter<BaseView>()
var preseter: Presenter<out BaseView> = Presenter<ProductView>()
preseter = productPresenter
```

类 A 和 B 包含了 preseter 变量（其中包含了变型修饰符）。这里暂时无法使用类型推断机制，最终导致代码较为冗长。为了对此加以改进，须在类型参数声明点处指定变型修饰符，如下所示：

```
interface BaseView
interface ProductView: BaseView
class Presenter<out T> // 1

// usage
// Variable declared inside class A and B

var preseter = Presenter<BaseView>()
var productPresenter = Presenter<ProductView>()
preseter = productPresenter
```

对于注释 1，变型修饰符在类型参数声明位置处定义了变型修饰符。

在 Presenter 类中，仅须定义变型修饰符一次。实际上，上述两种实现彼此等价，但声明位置变型更加简洁，同时还可被类的外部客户端方便地加以使用。

6.3.3 集合变型

在 Java 中，数组表示为协变数据。默认条件下，可传递 String[]数组，即使期望使用 Object[]类型的数组，如下所示：

```
public class Computer {
    public Computer() {
        String[] stringArray = new String[]{"a", "b", "c"};
        printArray(stringArray); // Pass instance of String[]
    }

    void printArray(Object[] array) {
         // Define parameter of type Object[]
         System.out.print(array);
    }
}
```

在 Java 的早期版本中，该操作行为十分重要，其原因在于：可使用不同的数组类型作为参数，如下所示：

```
// Java
static void print(Object[] array) {
    for (int i = 0; i <= array.length - 1; i++)
    System.out.print(array[i] + " ");
    System.out.println();
}

// Usage
String[] fruits = new String[] {"Pineapple","Apple", "Orange",
                                "Banana"};
print(fruits); // Prints: Pineapple Apple Orange Banana
Arrays.sort(fruits);
print(fruits); // Prints: Apple Banana Orange Pineapple
```

但该操作行为可导致潜在的运行期错误，如下所示：

```
public class Computer {
    public Computer() {
        Number[] numberArray = new Number[]{1, 2, 3};
        updateArray(numberArray);
    }
    void updateArray(Object[] array) {
        array[0] = "abc";
        // Error, java.lang.ArrayStoreException: java.lang.String
```

```
    }
}
```

其中，函数 updateArray 接收 Object[]类型的参数，且传递 String[]。同时，利用 String 参数调用 add 方法。由于数组项为 Object 类型，因而此处可使用 String，这也表示为一个新值。最后，还须添加 String 至仅包含 String 类型数据项的泛型数组中。考虑到默认的协变行为，编译器无法检测到这一问题，因而会抛出 ArrayStoreException 异常。

由于 Kotlin 编译器将此操作行为视为潜在危险，因而对应代码无法在 Kotlin 中被编译。这也体现了 Kotlin 中数组为不可变的原因。因此，当需要使用到 Array<Any>且传递非 Array<Number>时，将出现编译期错误，如下所示：

```
public class Array<T> { /*...*/ }
```

根据这一点，当需要使用 Array<Any>且传递非 Array<Number>类型时，将产生编译期错误，如下所示：

```
public class Array<T> { /*...*/ }
class Computer {
    init {
        val numberArray = arrayOf<Number>(1, 2, 3)
        updateArray(numberArray)
    }
    internal fun updateArray(array: Array<Any>) {
        array[0] = "abc"
        // error, java.lang.ArrayStoreException: java.lang.String
    }
}
```

注意，潜在的运行期异常仅发生于调整对象时。相应地，变型也应用于 Kotlin 集合接口中。在 Kotlin 标准库中，涵盖了采用两种不同方式定义的列表接口。其中，鉴于不可变特征，Kotlin List 接口具有协变特性（不包含任何可调整内部状态的方法）；而 Kotlin MutableList 则有所不同。下列代码显示了类型参数定义，如下所示：

```
interface List<out E> : Collection<E> { /*...*/ }
public interface MutableList<E> : List<E>, MutableCollection<E> {
    /*...*/
}
```

下面考察此类定义的实际操作结果，进而定义可变列表，以减少协变所带来的风险，如下所示：

```
fun addElement(mutableList: MutableList<Any>) {
    mutableList.add("Cat")
```

```
}

// Usage
val mutableIntList = mutableListOf(1, 2, 3, 4)
val mutableAnyList = mutableListOf<Any>(1, 'A')
addElement(mutableIntList) // Error: Type mismatch
addElement(mutableAnyList)
```

由于并不包含改变内部状态的方法，因而该列表处于安全状态，其协变行为可支持更为丰富的通用函数操作，如下所示：

```
fun printElements(list: List<Any>) {
    for(e in list) print(e)
}

// Usage
val intList = listOf(1, 2, 3, 4)
val anyList = listOf<Any>(1, 'A')
printElements(intList) // Prints: 1234
printElements(anyList) // Prints: 1A
```

此处，可向 printElements 函数传递 List<Any>或其子类型——List 接口具有协变特征。另外，由于 MutableList 接口的不可变性，仅可向 addElement 传递 MutableList<Any>。

当采用 in 和 out 修饰符时，可对变型行为予以操控。另外，还应注意变型包含了某些限制条件，下面对此加以讨论。

6.3.4 变型的生产者/消费者限制条件

通过变型修饰符，可针对类/接口（声明位置变型）或类型参数（使用位置变型）的特定类型参数获取协变/逆变特性。尽管如此，仍需要注意某些限制条件。为了确保安全性，Kotlin 编译器限制了类型参数的使用位置。

对于不可变特性（默认状态下在类型参数中不包含变型修饰符），可在 in（函数参数类型）和 out（函数返回类型）位置处使用类型参数，如下所示：

```
interface Stack<T> {
    fun push(t:T) // Generic type at in position
    fun pop():T // Generic type at out position
    fun swap(t:T):T // Generic type at in and out positions
    val last: T // Generic type at out position
    var special: T // Generic type at out position
}
```

当采用变型修饰符时，则仅限定于某个单一位置。这也表明，针对方法参数（in）或者方法返回值（out），仅可将类型参数用作某一类型。当前类可以定义为生产者或消费者，因而可以解释为：类接收参数或生成参数。

下面考察该限定条件与变型修饰符之间的关系如何在声明点处加以定义。针对两种类型参数 R 和 T，下列代码展示了其正确和错误的应用方式。

```
class ConsumerProducer<in T, out R> {
    fun consumeItemT(t: T): Unit { } // 1

    fun consumeItemR(r: R): Unit { } // 2, error
    fun produceItemT(): T { // 3, error
        // Return instance of type T
    }
    fun produceItemR(): R { // 4
        // Return instance of type R
    }
}
```

针对注释 1：正确，类型参数 TT 位于 in 位置处；对于注释 2：错误，类型参数 R 位于 in 位置处；对于注释 3：错误，类型参数 T 位于 out 位置处；对于注释 4：正确，类型参数 R 位于 out 位置处。

不难发现，若当前配置被禁用，编译器将报告一条错误信息。注意，可针对两种类型参数 R 和 T 添加不同的修饰符。

位置限定条件仅适用于类外部可访问（可见）的方法，这也表明，对应方法不仅包括之前所用的全部 public 方法（public 定义为默认的标识符），而且还涵盖了标记为 protected 或 internal 的方法。若将方法的可见性修改为 private，则可在任意位置使用类型参数（R 和 T），类似于不可变的类型参数，如下所示：

```
class ConsumerProducer<in T, out R> {
    private fun consumeItemT(t: T): Unit { }
    private fun consumeItemR(r: R): Unit { }
    private fun produceItemT(): T {
        // Return instance of type T
    }
    private fun produceItemR(): R {
        //Return instance of type R
    }
}
```

对于用作类型的类型参数，表 6.1 显示了全部所允许的位置。

表 6.1

可见性修饰符	不可变性	协变性（out）	逆变性（in）
public，protected，internal	in/out	out	in
private	in/out	in/out	in/out

6.3.5　不可变构造函数

对于之前描述的 in 和 out 位置规则，存在一个较为重要的例外情形：构造函数参数通常处于不可变状态，如下所示：

```
class Producer<out T>(t: T)
// Usage
val stringProducer = Producer("A")
val anyProducer: Producer<Any> = stringProducer
```

其中，构造函数表示为公有类型，类型参数 T 则声明为 out，但在 in 位置处，仍可将其用作构造函数参数类型。其原因在于，构造函数方法无法在实例创建后被调用，因而一直处于安全调用状态。

第 4 章曾讨论到，可通过 val 或 var 修饰符直接在类构造函数中定义属性。当指定协变特性时，可在包含协变类型的构造函数中仅定义只读（val）属性，因而处于安全状态，其原因在于：仅生成 get 方法，因而该属性值无法在类实例化后被修改，如下所示：

```
class Producer<out T>(val t: T) // Ok, safe
```

当使用 var 时，get 和 set 方法均由编译器生成，因此，属性值可在某点处产生变化。这也解释了为何无法在构造函数中声明一个协变类型的只写（var）属性，如下所示：

```
class Producer<out T>(var t: T) // Error, not safe
```

如前所述，变型限定条件仅适用于外部客户端，因而仍可定义一个只写属性，即添加一个私有可见性标识符，如下所示：

```
class Producer<out T>(private var t:T)
```

另一个较为常见的泛型限制条件则与类型擦除有关（源自 Java）。

6.4　类 型 擦 除

JVM 中引入了类型擦除，以使 JVM 字节码向后兼容于泛型出现之前的某些版本。在

Android 平台上，Kotlin 和 Java 均编译为 JVM 字节码，因而都会受到类型擦除的影响。

类型擦除是指从泛型中移除类型参数这一处理过程，因而泛型在运行期时将丢失某些类型信息（类型参数），如下所示：

```
package test
class Box<T>

val intBox = Box<Int>()
val stringBox = Box<String>()

println(intBox.javaClass) // prints: test.Box
println(stringBox.javaClass) // prints: test.Box
```

编译器可区分两种类型，并可确保类型安全性。然而，在编译时，参数化类型 Box<Int>和 Box<String>均被编译器转换为 Box（原始类型）。生成的 Java 字节码并不包含与类型参数相关的任何信息，因而无法在运行期区分泛型。

类型擦除可能会导致某些问题的出现。在 JVM 中，无法声明相同方法的重载版本（包含相同的 JVM 签名），如下所示：

```
/*
java.lang.ClassFormatError: Duplicate method name&signature...
*/
fun sum(ints: List<Int>) {
    println("Ints")
}

fun sum(strings: List<String>) {
    println("Ints")
}
```

当移除类型参数后，上述两个方法将包含相同的声明，如下所示：

```
/*
java.lang.ClassFormatError: Duplicate method name&signature...
*/
fun sum(ints: List) {
    println("Ints")
}
fun sum(strings: List) {
    println("Ints")
}
```

除此之外，还可通过修改生成的函数 JVM 名来解决这一类问题。对此，当代码编译为 JVM 字节码时，可使用 JvmName 修改某个方法名，如下所示：

```
@JvmName("intSum") fun sum(ints: List<Int>) {
    println("Ints")
}
fun sum(strings: List<String>) {
    println("Ints")
}
```

在 Kotlin 中，函数在使用过程中未发生任何变化，但由于修改了第一个函数的 JVM 名，因而需要通过新名称在 Java 中使用该函数，如下所示：

```
// Java
TestKt.intSum(listOfInts);
```

某些时候，可能需要在运行期保留类型参数，对此，可使用 reified 类型参数。

6.4.1　reified 类型参数

在某些场合下，在运行期访问类型参数十分有用，但考虑到类型擦除，该操作往往被禁止，如下所示：

```
fun <T> typeCheck(s: Any) {
    if(s is T){
    // Error: cannot check for instance of erased type: T
        println("The same types")
    } else {
        println("Different types")
    }
}
```

为了克服 JVM 限制条件，Kotlin 采用了特定修饰符，并可在运行期持有类型参数。对此，需要通过 reified 标识符标记类型参数，如下所示：

```
interface View
class ProfileView: View
class HomeView: View
inline fun <reified T> typeCheck(s: Any) { // 1
    if(s is T){
        println("The same types")
    } else {
    println("Different types")
```

```
    }
}
// Usage
typeCheck<ProfileView>(ProfileView()) // Prints: The same types
typeCheck<HomeView>(ProfileView()) // Prints: Different types
typeCheck<View>(ProfileView()) // Prints: The same types
```

在注释 1 中，显示了标记为 reified 的类型参数，以及标记为 inline 的函数。

当前，可在运行期内安全地访问类型参数。其中，reified 类型参数仅与内联函数协同工作——在编译（内联）期间，Kotlin 编译器将替换 reified 类型参数的实际类。通过这一方式，类型参数不会被类型擦除所移除。

另外，还可在 reified 类型上使用反射机制，进而获得与当前类型相关的更多信息，如下所示：

```
inline fun <reified T> isOpen(): Boolean {
    return T::class.isOpen
}
```

reified 类型参数在 JVM 字节码级别中有所体现，并针对基础类型作为一种真实类型或封装器类型，因而 reified 类型参数不会受到类型擦除的影响。

reified 类型参数可通过一种全新的方法编写代码。当在 Java 中启动一个新的 Activity 时，可编写如下代码：

```
// Java
startActivity(Intent(this, ProductActivity::class.java))
```

在 Kotlin 中，可定义 startActivity 方法，并通过相对简单的方式导航至 Activity，如下所示：

```
inline fun <reified T : Activity> startActivity(context: Context) {
    context.startActivity(Intent(context, T::class.java))
}

// Usage
startActivity<UserDetailsActivity>(context)
```

之前定义了 startActivity 方法，通过类型参数，可传递与 Activity（ProductActivity）相关的信息。另外，还须定义一个显式的 reified 类型参数约束，以确保仅可使用 Activity（及其子类）作为类型参数。

6.4.2 startActivity 方法

为了正确使用 startActivity 方法，需要一种方式将参数传递至启动的 Activity（Bundle）

中。相应地，可能需要更新之前的实现，以支持下列参数：

```
startActivity<ProductActivity>("id" to 123, "extended" to true)
```

在上述示例中，参数通过所提供的键值对被过滤（定义于 to 内联函数中）。该函数的具体实现则超出了本书的讨论范围。尽管如此，仍可使用现有的函数。其中，Anko 库（位于 https:/ /github. com/ Kotlin/anko）已实现了 startActivity 方法，并包含了全部所需功能。此处仅须导入 Appcompat-v7-commons 即可，如下所示：

```
compile "org.jetbrains.anko:anko-appcompat-v7-commons:$anko_version"
```

Anko 针对 Context 和 Fragment 类定义了扩展，因而可像类中的其他方法那样，使用 Activity 或 Fragment 中的当前方法，且无须再在前类中定义方法。关于扩展，第 7 章将对此加以讨论。

需要注意的是，reified 类型参数包含了一个主要的限制条件：无法从 reified 类型参数中创建类实例（不使用反射机制），其后的原因可描述为：构造函数通常仅与实例（一般不采用继承）关联，因而不存在适用于所有类型参数的构造函数。

6.5　星号投射

鉴于类型擦除，运行期内往往会出现不完整的类型信息。例如，无法得到泛型类型参数，如下所示：

```
val list = listOf(1,2,3)
println(list.javaClass) // Prints: class java.util.Arrays$ArrayList
```

这将会产生某些问题。例如，无法执行任何检测，以验证 List 包含的数据元素类型，如下所示：

```
/*
Compile time error: cannot check instance of erased type:
List<String>
*/
if(collection is List<Int>) {
    //...
}
```

由于检测在运行期内执行，且与类型参数相关的信息尚不完整，因而会出现此类问题。然而，与 Java 中的操作相反，Kotlin 不允许声明原始类型（未利用类型参数实现参数化的泛型），如下所示：

```
SimpleList<> // Java: ok
SimpleList<> // Kotlin: error
```

相反，Kotlin 使用星号投射语法，基本上，与类型参数相关的信息缺失或者不太重要，如下所示：

```
if(collection is List<*>) {
    //...
}
```

通过星号投射语法，可以说 Box 存储了特定类型的参数，如下所示：

```
class Box<T>

val anyBox = Box<Any>()
val intBox = Box<Int>()
val stringBox = Box<String>()
var unknownBox: Box<*>

unknownBox = anyBox // Ok
unknownBox = intBox // Ok
unknownBox = stringBox // Ok
```

需要注意的是，Box<*>和 Box<Any>之间稍有不同。如果希望定义包含 Any 数据项的列表，则可使用 Box<Any>；然而，如果打算定义包含特定类型数据项的列表，但该类型未知（可能是 Any、Int、String 等，且并不包含与该类型相关的信息），此时 Box<Any>意味着当前列表包含了 Any 类型列表。对此，可使用 Box<*>，如下所示：

```
val anyBox: Box<Any> = Box<Int> // Error: Type mismatch
```

如果某个泛型利用多个类型参数加以定义，则需要针对缺失的类型参数使用星号（*），如下所示：

```
class Container<T, T2>
val container: Container<*, *>
```

当需要在该类型上执行某项操作时，星号投射十分有用，而与类型参数相关的信息则显得不那么重要，如下所示：

```
fun printSize(list: MutableList<*>) {
    println(list.size)
}
```

```
// usage
val stringList = mutableListOf("5", "a", "2", "d")
val intList = mutableListOf(3, 7)
printSize(stringList) // prints: 4
printSize(intList) // prints: 2
```

在上述示例中，并不需要使用到与类型参数相关的信息以确定集合的尺寸。如果不使用取决于类型参数的相关方法，星号投射语法可减少 variance 修饰符的使用概率。

6.6 类型参数命名规则

针对参数的命名，官方发布的 Java 类型参数命名规则（参见 https://docs.oracle.com/javase/tutorial/java/generics/types.html）定义了以下指导原则：

默认状态下，类型参数名称表示为大写的单一字母，这与已知的变量命名规则形成了鲜明的对比。如果缺少这一点，将难以区分类型变量和常规类（或接口）之间的差别。其中，较为常见的类型参数名称如下所示。

- E：数据元素（广泛地用于 Java 集合框架中）。
- K：键。
- N：数字。
- T：类型。
- V：值。
- S、U、V 等：表示为第二种、第三种、第四种类型。

Kotlin 标准库中的许多类均支持这一规则，且针对常见类均工作良好，例如普通的类（List、Mat、Set 等），或者定义了某个简单类型参数的类（Box<T>类）。然而，当使用自定义类以及多个类型参数时，读者会迅速意识到，单字母无法包含足够的信息量，某些时候，将难以了解类型参数表达的数据类型。针对这一问题，存在多种处理方案。

另外，应确保泛型实现较好的文档化管理。尽管如此，通过简单地查看代码，有时仍无法确定类型参数的真实含义。虽然文档十分重要，但仍然是一类辅助信息源，读者还是应提升代码的可读性。

多年以来，程序员已倾向于遵循具有实际意义的命名规则。Google Java 风格指南（参见 https://google.github.io/styleguide/javaguide.html#s5.2.8- type-variable-names）简要地介绍了官方类型参数命名规则以及自定义命名规则之间的混合操作，并提出了两种独特的风格。第一种风格是使用单一的大写字母，随后是可选的单一数字（不同于 Java 中描述的 S、U、V 名称），如下所示：

```
class Box<T, T2>
```

第二种风格则更具描述性，并针对类型参数添加了富有含义的前缀，如下所示：

```
class Box<RequestT>
```

然而，针对类型参数名称，目前尚不存在单一标准。较为常见的解决方案是使用单一大写字母。需要注意的是，类一般常使用泛型，因而适宜的命名机制将改善代码的可读性。

6.7 本章小结

本章讨论了泛型这一概念，泛型类和接口的定义方式，以及泛型的声明方式。通过使用点和声明点 variance 修饰符，本章介绍了子类型关系的处理方式。另外，本章还解释了类型擦除，以及如何通过 reified 类型参数在运行期内持有泛型。

第 7 章将讨论 Kotlin 中较为重要的特性之一，即扩展，该特性可向现有类中添加新的操作行为。此外，还将探讨如何针对任意既定类实现相应的新方法和属性，包括源自 Android 框架和第三方库的结果类。

第 7 章　扩展函数和属性

在第 6 章中，相信大多数开发人员对其中所介绍的概念已十分熟悉。本章主要考察 Java 中未曾涉及的内容——扩展。这也是 Kotlin 中较为优异的特性，且受到了大量 Kotlin 程序员的喜爱。在 Android 开发中，扩展可视为一项重大的改进措施。

本章主要涉及以下内容：

- 扩展函数。
- 扩展属性。
- 成员扩展函数。
- 泛型扩展函数。
- 集合处理。
- 包含接收者的函数类型，以及包含接收者的函数字面值。
- 面向任意对象的 Kotlin 泛型扩展函数。
- Kotlin 域特定语言。

7.1　扩 展 函 数

所有大型 Java 项目一般均会使用到工具类，例如 StringUtils、ListUtils、AndroidUtils 等。这一类工具函数具有普遍模式，并采用相对简单的方式测试和使用。相关问题主要体现在，Java 对于此类函数的创建和应用的支持度较差，其原因在于：此类函数须实现为某个类的静态函数。下面通过一个示例讨论这一问题。Java Android 开发人员基本都熟悉下列 Toast 输出代码：

```
Toast.makeText(context, text, Toast.LENGTH_SHORT).show();
```

当显示错误或短消息时，常会在 Android 项目中对此加以使用；同时，在大多数 Android 教程中，一般会在开始处即予以介绍。由于采用类似于构造器的方式使用静态函数，因而实现代码较为冗长。或许，Java Android 开发人员或多或少会在返回对象上忘记调用 show 方法，进而检测全部环境条件以寻找其中的原因。对此，有必要将这一简单函数封装为一个工具函数。但实际上，通常较少使用这一方式，为了理解这一点，考察 Java 中的下列实现方式：

```
public class AndroidUtils {
   public static void toast(Context context, String text) {
```

```
        Toast.makeText(context, text, Toast.LENGTH SHORT).show();
    }
}

// Usage
AndroidUtils.toast(context, "Some toast");
```

当程序员需要使用下列函数时，需要回忆是否存在该函数、类的位置以及名称。因此，其应用并未得到简化。如果未修改 Android SDK 实现，一般不可能将其实现为 Context 方法（Activity 的子类）。但在 Kotlin 中，则可以生成扩展函数，其行为类似于定义于类中的实际方法。下列代码显示了作为 Context 扩展的 toast 的实现方式：

```
fun Context.toast(text: String) { // 1
    Toast.makeText(this, text, LENGTH LONG).show() //2
}

// Usage
context.toast("Some toast")
```

针对注释 1，Context 并未出现于参数列表中，而是位于函数名之前，这体现了扩展类型的定义方式；对于注释 2，在函数体内部，可使用 this 关键字，进而引用相关对象（扩展函数于其上被调用）。

扩展函数和标准函数之间的解构差异在于，函数名称之前存在一个接收者类型。另外，函数体内部也包含少许可见性变化——其中，可通过 this 关键字访问接收者对象（在该对象上，扩展将被调用）；或者，也可直接调用其函数或属性。根据这一定义，toast 函数的行为类似于定义于 Context 中的一个方法，如下所示：

```
context.toast("Some toast")

Alternatively:
class MainActivity :Activity() {
    override fun onCreate(savedInstanceState: Bundle?){
        super.onCreate(savedInstanceState)
        toast("Some text")
   }
}
```

相比于 toast 显示代码的全部实现，这将使得 toast 的使用更加方便。另外，还可获得源自 IDE 的提示信息，并在位于 Context 内部（类似于 Activity 内部）或 Context 实例上时调用该函数，如图 7.1 和图 7.2 所示。

```
class MainActivity : Activity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        toa
        toast(text: String) for Context in com.naxtlevelofandroiddevelopme… Uni
}   Press Ctrl+Period to choose the selected (or first) suggestion and insert a dot afterwards >>
```

图 7.1

```
context.toa
    toast(text: String) for Context in com.naxtlevelofandroiddevelopme… Uni
    Press Ctrl+Period to choose the selected (or first) suggestion and insert a dot afterwards >>
```

图 7.2

在上述示例中，Context 表示为 toast 函数的接收者类型，this 实例则表示为接收者对象的引用。接收者对象的全部函数和属性均可显式地予以访问，因而有下列定义：

```
fun Collection<Int>.dropPercent(percent: Double)
    = this.drop(floor(this.size * percent)
```

随后，可利用下列代码进行替换：

```
fun Collection<Int>.dropPercent(percent: Double)
    = drop(floor(size * percent))
```

扩展函数的应用体现在多方面。类似的扩展函数也可针对 View、List、String、定义于 Android 框架或第三方库中的其他类，以及开发人员的自定义类加以定义。扩展函数还可添加至任意可访问类型，甚至是 Any 对象。下列代码显示了可在各个对象上调用的扩展函数：

```
fun Any?.logError(error: Throwable, message: String = "error") {
    Log.e(this?.javaClass?.simpleName ?: "null", message, error)
}
```

下列代码显示了调用示例：

```
user.logError(e, "NameError") // Logs: User: NameError ...
"String".logError(e) // String: error ...
logError(e) // 1, MainActivity: error ...
```

针对注释 1，在 MainActivity 中进行调用。

7.1.1　扩展函数底层机制

Kotlin 函数貌似复杂，实际上，其底层机制十分简单。基于第一个参数的接收者对象，

顶级扩展函数编译为静态函数。下面再次考察之前的 toast 函数，如下所示：

```
// ContextExt.kt
fun Context.toast(text: String) {
    Toast.makeText(this, text, LENGTH_LONG).show()
}
```

在编译并反编译至 Java 后，该函数如下所示：

```
// Java
public class ContextExtKt {
    public static void toast(Context receiver, String text) {
        Toast.makeText(receiver, text, Toast.LENGTH_SHORT).show();
    }
}
```

根据第一个参数上的接收者对象，Kotlin 顶级扩展函数编译为静态函数，这也是依然使用 Java 扩展的原因，如下所示：

```
// Java
ContextExtKt.toast(context, "Some toast")
```

除此之外，从 JVM 字节码的角度来看，这也意味着，当前方法并非真正被添加。但在编译期，所有扩展函数引用均被编译为静态函数调用。当扩展函数仅表示为一类函数时，函数修饰符仍可应用于其上，并与其他函数的应用方式相同。例如，扩展函数可标记为 inline，如下所示：

```
inline fun Context.isPermissionGranted (permission: String): Boolean =
ContextCompat.checkSelfPermission (this, permission) ==
PackageManager.PERMISSION_GRANTED
```

类似于其他 inline 函数，该函数调用在应用程序编译期时，将被实际的函数体替换。在实际操作过程中，扩展函数与其他函数的处理方式并无两样，例如，可表示为独立表达式、包含默认参数、通过命名参数加以使用，等等。但此类实现缺少应有的直观结果，下面将对此加以讨论。

1. 不存在重载方法

若成员函数和扩展函数具有相同的名称和参数，则优先选择成员函数，如下所示：

```
class A {
    fun foo() {
        println("foo from A")
    }
```

```
}

fun A.foo() {
    println("foo from Extension")
}

A().foo() // Prints: foo from A
```

上述代码可正常工作。另外，即使是超类中的方法也将优于扩展函数，如下所示：

```
open class A {
    fun foo() {
        println("foo from A")
    }
}

class B: A()

fun B.foo() {
    println("foo from Extension")
}
A().foo() // foo from A
```

据此，扩展函数不允许修改实际对象的行为，且仅可添加附加功能。由于无法调整所用对象的操作行为（否则，可能导致错误且难以跟踪），因而可确保安全性。

2. 访问接收者元素

根据第一个函数上的接收者对象，扩展函数将编译为静态函数，因而不具备额外的访问权限。private 和 protected 数据元素均不可被访问，而包含 Java default、Java package 或 Kotlin internal 修饰符的元素，其访问方式与其他标准对象一致。

因此，此类数据元素得到了应有的保护。注意，尽管扩展函数功能强大且十分有用，但仍是一类语法糖，且并无太多新奇之处。

3. 采用静态方式处理扩展

扩展函数仅表示为包含接收者（作为第一个参数）的函数，因此，其调用在编译期内由调用函数的类型处理。例如，若针对超类和子类定义了扩展，则在调用期间选取的扩展函数则取决于所操作的属性类型，如下所示：

```
abstract class A
class B: A()
```

```
fun A.foo() { println("foo(A)") }
fun B.foo() { println("foo(B)") }

val b = B()
b.foo() // prints: foo(B)
(b as A).foo() // 1, prints: foo(A)
val a: A = b
a.foo() // 1, prints: foo(A)
```

针对注释 1，此处希望 foo(B)为类型 B，但由于扩展采用静态方式加以处理，因而针对 A 使用扩展函数——变量为类型 A，关于编译期内存在何种对象，当前无法获取相关信息。

某些时候，当我们将扩展函数定义为最常被转换的类型时，不应该将扩展函数实现到它的子类。

这是一个较为重要的限定条件，读者应对此牢记，尤其是在公有库实现中。否则，某些扩展函数将阻碍其他函数，并产生难以预料的错误行为。

7.1.2　伴生对象扩展

如果某个类包含所定义的伴生对象，那么也可针对该伴生对象定义扩展函数（以及属性）。为了进一步区分某个类的扩展，以及伴生对象的扩展，需要在扩展类型和函数名之间添加.Companion，如下所示：

```
class A {
    companion object {}
}
fun A.Companion.foo() { print(2) }
```

待定义完毕后，foo 方法可像在 A 伴生对象中定义那样予以使用，如下所示：

```
A.foo()
```

需要注意的是，此处采用类这一类型调用扩展，而非类实例。如果针对某个伴生对象可创建扩展函数，则需要在当前类显式地定义一个伴生对象，即使是一个空对象；否则将无法定义扩展函数，如图 7.3 所示。

```
class A {}

fun A.Companion.foo() { print(2) }
```

Unresolved reference: Companion

图 7.3

7.1.3　通过扩展函数重载操作符

操作符重载是 Kotlin 中的一个重要特性，但需要使用到 Java 库或非本地操作符。例如，在 RxJava 中，采用 CompositeDisposable 函数管理订阅（subscription）。该集合使用 add 方法添加新的元素。下列代码显示了添加至 CompositeDisposable 的订阅示例：

```
val subscriptions = CompositeDisposable()

subscriptions.add(repository
    .getAllCharacters(qualifiedSearchQuery)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(this::charactersLoaded, view::showError))
```

对于向可变集合中添加新元素，Kotlin 的标准方式则是使用 plusAssign（+=）操作符。该方法不仅应用范围较广，而且具备简洁特征，同时还可忽略括号，如下所示：

```
val list = mutableListOf(1,2,3)
list.add(1)
list += 1
```

对于当前示例中的应用，可添加下列扩展：

```
operator fun CompositeDisposable.plusAssign(disposable: Disposable)
{
    add(disposable)
}
```

随后，即可在 CompositeDisposable 上使用 plusAssign 方法，如下所示：

```
subscriptions += repository
    .getAllCharacters(qualifiedSearchQuery)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(this::charactersLoaded, view::showError)
```

7.1.4　顶级函数的应用位置

当发觉其他程序员定义的类中缺少某个方法，较为常见的做法是使用扩展函数。例如，当确定 View 应添加 show 和 hide 方法时，与设置可见字段相比，扩展函数更加简单，并可采用手动方式完成，如下所示：

```
fun View.show() { visibility = View.VISIBLE }
fun View.hide() { visibility = View.GONE }
```

此处无须记住加载 util 函数的类名。在 IDE 中，仅须在当前对象之后输入“.”，即可搜索当前项目和库中的方法连同其对象扩展，并使其看起来像是原始对象成员。虽然这可视作扩展函数的优点，但其中也包含了某些危险。目前已存在大量的、表示为扩展函数包的 Kotlin 库。当使用多个扩展函数时，Android 代码则与常规代码有所差异。当然，该方式有利有弊，其优点包括：

- 代码短小且具有可读性。
- 代码体现了相关逻辑，而非 Android 样板文件。
- 扩展函数常被用于测试；或者，至少可在多处加以使用。因此，可方便地了解到扩展函数是否处于正常工作状态。
- 当使用扩展函数时，出现差错的概率也大大降低；否则，代码调试工作将十分耗时。关于最后两点内容，下面再次考察 toast 函数。当采用下列方式编写代码时，一般很难出现错误。

```
toast("Some text")
```

相比之下，下列方式则易于产生错误：

```
Toast.makeText(this, "Some text",Toast.LENGTH_LONG).show()
```

在项目中使用大量的扩展应用，其主要问题在于：实际上，我们正在制定自己的 API、命名和实现函数、确定参数内容。当某位开发人员加入到团队中时，他需要学习所创建的全部 API 内容。尽管 Android API 包含不少缺陷，但其优势主要体现在应用广泛，且为大多数 Android 开发人员所知。

这是否意味着我们需要放弃扩展？当然不是！扩展是一项十分有用的特性，可帮助我们使代码更加短小精悍，同时提高代码的简洁性。据此，应采取更加智能的方式使用扩展，其中包括：

- 避免出现实现同一任务的多个扩展。
- 一些简短、简单的功能一般不需要使用到扩展。
- 项目中保持一种代码风格。对此，应与团队进行交流并制定某些标准。
- 当使用包含扩展的共有库时，应予以谨慎处理：不可修改其中的代码，所编写的扩展应与其匹配，以保持 API 的简洁性。

7.2　扩 展 属 性

本节首先讨论扩展属性的定义，随后学习此类属性的可用位置。如前所述，Kotlin 的

属性通过其访问器（getter 和 setter）加以定义，如下所示：

```
class User(val name: String, val surname: String) {
    val fullName: String
    get() = "$name $surname"
}
```

除此之外，还可定义扩展属性，唯一的限制条件是，该属性不可包含幕后字段，其原因在于，扩展无法存储状态，因而不存在适宜的位置可存储该字段。下列代码针对 TextView 定义了扩展字段。

```
val TextView.trimmedText: String
get() = text.toString().trim()

// Usage
textView.trimmedText
```

类似于扩展函数，上述实现将作为访问器函数（基于第一个参数上的接收者）被编译。下列代码显示了 Java 中的简化结果。

```
public class AndroidUtilsKt {
    String getTrimmedText(TextView receiver) {
        return receiver.getText().toString().trim();
    }
}
```

如果定义为读–写属性，setter 和 getter 都需要予以实现。回忆一下，仅不需要 Java 字段的属性允许定义为扩展属性。例如，图 7.4 和图 7.5 中的代码属于“非法”内容。

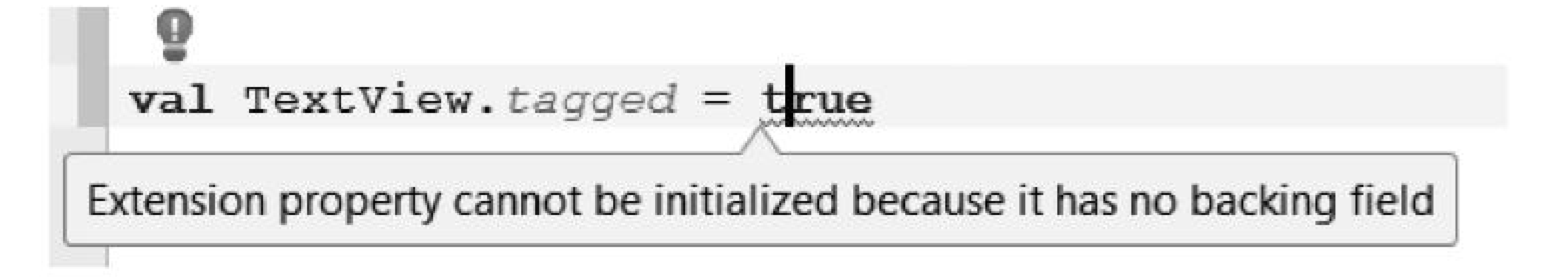


图 7.4

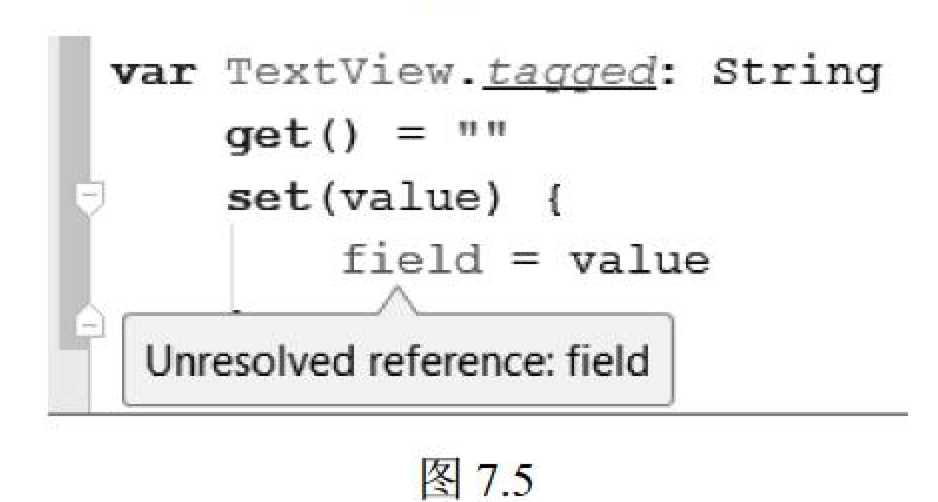


图 7.5

扩展属性可与扩展函数实现交互使用，且常用作顶级工具函数。当需要某个对象包含一些非本地开发属性时，可使用扩展属性。对于扩展函数和扩展属性，其使用决策类似于使用函数或属性（类中不包含幕后字段）。作为提示，当底层算法满足下列条件时，根据规则应优先使用属性，而非函数。

- 算法未抛出错误。
- 算法复杂度为 O(1)。
- 算法计算量较小（或者在第一轮中被缓存）。
- 多次调用返回相同的结果。

下面考察一个简单的问题。一般情况下，开发人员需要使用到 Android 中的某些服务，但所用代码可能比较复杂，如下所示：

```
PreferenceManager.getDefaultSharedPreferences(this)
getSystemService(Context.LAYOUT_INFLATER_SERVICE) as LayoutInflater
getSystemService(Context.ALARM_SERVICE) as AlarmManager
```

当使用 AlarmManager 或 LayoutInflater 这一类服务时，程序员须了解以下内容：

- 提供服务的函数名(例如 getSystemService)，以及包含该函数的类名(例如 Context)。
- 指定该服务的字段名（例如 Context.ALARM_SERVICE）。
- 当前服务应转换的类名（例如 AlarmManager）。

上述过程较为复杂，因而可通过扩展属性对应用进行优化。对此，可采用下列方式定义扩展属性：

```
val Context.preferences: SharedPreferences
    get() = PreferenceManager
        .getDefaultSharedPreferences(this)

val Context.inflater: LayoutInflater
    get() = getSystemService(Context.LAYOUT_INFLATER_SERVICE)
        as LayoutInflater

val Context.alarmManager: AlarmManager
    get() = getSystemService(Context.ALARM_SERVICE)
        as AlarmManager
```

自此，可作为 Context 属性使用 preferences、inflater 和 alarmManager，如下所示：

```
context.preferences.contains("Some Key")
context.inflater.inflate(R.layout.activity_main, root)
context.alarmManager.setRepeating(ELAPSED_REALTIME, triggerAt,
   interval, pendingIntent)
```

上述示例展示了只读扩展函数的应用，下面重点讨论 inflater 扩展属性，该属性可帮助获取常用但在缺少扩展时难以获取的元素。其有效性体现在，程序员仅须记住：所需内容仅为一个 inflater，且需要 Context 对其加以确定。同时，程序员无须记忆提供系统服务的方法名（getSystemService）及其所处位置（Context），以及该服务应转换的类型（AlarmManager）。换而言之，这一扩展节省了大量的工作时间以及内存空间。另外，属性 getter 的执行时间较短，其复杂度为 O(1)，且不会抛出任何错误，同时还将返回相同的 inflater（实际上是不同的实例，但从程序员角度来看，其应用通常保持一致，这一点十分重要）。

前述内容讨论了只读扩展属性，但尚未考察只写扩展属性。在下面的示例中，此类属性可替代之前的 hide 和 show 方法。

```
var View.visible: Boolean
get() = visibility == View.VISIBLE
set(value) {
    visibility = if (value) View.VISIBLE else View.GONE
}
```

通过下列属性，可修改视图元素的可见性：

```
button.visible = true // the same as show()
button.visible = false // the same as hide()
```

除此之外，还可进一步检测视图元素的可见性，如下所示：

```
if(button.visible) { /* ... */ }
```

当定义完毕后，可将其视作一个 View 属性。另外重要的一点是，所设置的内容与获取的内容保持一致。因此，假设不存在修改元素可见性的其他线程，则可设置某些属性值，如下所示：

```
view.visible = true
```

随后，getter 将提供相同值，如下所示：

```
println(view.visible) // Prints: true
```

最后，getter 和 setter 中不存在其他逻辑——仅涉及特定属性中的变化内容，因而也可满足之前描述的其他各项规则。

7.3　成员扩展函数和属性

前述内容讨论了顶级（top-level）函数和属性，但也可在某个类或对象中对其加以定义，其中定义的扩展称作成员扩展，与顶级扩展相比，常用于不同类型的问题。

下面查看最为简单的示例，其中使用了成员扩展。假设需要删除 String 列表中各个“第 3 个”元素，下列扩展函数支持每个“第 i 个”元素的删除操作：

```
fun List<String>.dropOneEvery(i: Int) =
    filterIndexed { index, _ -> index % i == (i - 1) }
```

该函数的问题在于，不应作为一个工具扩展被析取，其原因在于：

- 不适用于不同的列表类型（例如 User 或 Int 列表）。
- 应用范围狭窄，且不会用于当前项目中的其他处。

这也是将其定义为 private 的原因。较好的方法是作为成员扩展函数，将其置于当前所用类的内部，如下所示：

```
class UsersItemAdapter : ItemAdapter() {
    lateinit var usersNames: List<String>

    fun processList() {
    usersNames = getUsersList()
        .map { it.name }
        .dropOneEvery(3)
    }

    fun List<String>.dropOneEvery(i: Int) =
        filterIndexed { index,    -> index % i == (i - 1) }
    // ...
}
```

这也可视作使用成员扩展函数的第一个原因，并以此保护函数的可访问性。其中，可在同一文件中的最上方顶级位置处定义一个函数，并使用 private 修饰符。但成员扩展函数的行为与顶级函数有所不同。在前述代码中，此类函数使用了 public 修饰符，且仅可在 List <String>上以及 UsersItemAdapter 中被调用。因此，函数仅可在 UsersItemAdapter 及其子类，或者 UsersItemAdapter 的扩展函数中加以使用，如下所示：

```
fun UsersItemAdapter.updateUserList(newUsers: List<User>) {
    usersNames = newUsers
       .map { it.name }
       .dropOneEvery(3)
}
```

需要注意的是，当使用成员扩展函数时，需要使用到实现该函数的对象，以及该扩展函数被调用的对象，其原因在于，可使用两个对象中的元素。这可视作与成员扩展相关的重要信息：可以使用源自接收者类型和成员类型（不包含限定符）的元素。下面考察其应用方式，该示例使用了私有属性 category，如下所示：

```
class UsersItemAdapter(
    private val category: Category
) : ItemAdapter() {

   lateinit var usersNames: List<String>

   fun processList() {
        usersNames = getUsersList()
            .fromSameCategory()
            .map { it.name }
    }

    fun List<User>.fromSameCategory() =
        filter { u -> u.category.id == category.id }

    private fun getUsersList() = emptyList<User>()
}
```

在成员扩展函数 fromSameCategory 中，代码在扩展接收器（List<User>）进行操作，但同时也使用了源自 UsersItemAdapter 的 category 属性。不难发现，通过这一方式定义的函数需要使用到一个方法，并可采用与其他方法类似的方式加以使用。标准方法所蕴含的优势在于，可在 List 上调用，因而提供简洁的流式处理，而非非扩展方法应用，如下所示：

```
// fromSameCategory defined as standard method
usersNames = fromSameCategory(newUsers)
     .dropLast(3)

// fromSameCategory defined as member extension function
usersNames = newUsers
     .fromSameCategory()
     .dropLast(3)
```

另一种较为常见的应用是，成员扩展函数或属性可像常规方法那样加以使用，但需要基于以下事实：在成员函数内部，可使用接收者属性和方法，且无须对其加以命名。当采用这一方式时，可包含较少的语法；另外，还可在接收者上对其加以调用，而不是使用与参数相同的类型对其进行调用。作为示例，对应方法如下所示：

```
private fun setUpRecyclerView(recyclerView: RecyclerView) {
    recyclerView.layoutManager
        = LinearLayoutManager(recyclerView.context)
    recyclerView.adapter
        = MessagesAdapter(mutableListOf())
```

```
}

// Usage
setUpRecyclerView(recyclerView)
```

随后，可利用下列成员扩展函数对其替换：

```
private fun RecyclerView.setUp() {
    layoutManager = LinearLayoutManager(context)
    adapter = MessagesAdapter(mutableListOf())
}

// Usage
recyclerView.setUp()
```

当使用成员扩展函数时，可实现简单的调用以及函数体。该方案的最大问题在于，无法清晰地描述所采用的哪一个函数为 RecyclerView 的成员，哪一个函数表示为 Activity 和 RecyclerView 扩展的成员。稍后将对其加以分析。

7.3.1　接收者类型

当定义了成员扩展函数后，管理调用元素将变得更加复杂。在成员函数内，可隐式地访问下列内容：

- 源自该类和属性的成员函数和属性。
- 源自接收者类型及其超类的接收者类型函数和属性。
- 顶级函数和属性。

因此，在 setUp 扩展函数内部，可使用成员和接收者方法和属性，如下所示：

```
class MainActivity: Activity() {

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContentView(R.layout.main activity)
       val buttonView = findViewById(R.id.button view) as Button
       buttonView.setUp()
   }

   private fun Button.setUp() {
       setText("Click me!") // 1, 2
       setOnClickListener { showText("Hello") } // 2
   }
   private fun showText(text: String) {
```

```
        toast(text)
    }
}
```

针对注释 1，setText 表示为 Button 类方法；对于注释 2，可交替使用 Button 类和 MainActivity 类成员。

这其中涉及某些技巧——可能大多数人并不会意识到是否会存在错误；另外，setText 调用可能会被 showText 调用所交换。

虽然可在不同的接收者以及成员扩展元素内部加以使用，为了对此予以区分，各种接收者均已被命名。首先，使用此类关键字的所有对象称作隐式接收者，表示为成员并可在不使用限定符的情况下被访问，在 setUp 函数内部，存在两个隐式接收者，如下所示。

- 扩展接收者：针对 Button 所定义的扩展的类实例。
- 分发接收者：表示为类实例，扩展于其中被声明（MainActivity）。

需要注意的是，扩展接收者和分发接收者的成员均为同一体内的隐式成员，一种可能的情况是，使用了两个接收者中包含相同签名的成员。例如，如果将上述类修改为在 textView 中显示文本，而不是在 toast 函数中对其予以显示，同时将方法名修改为 setText，这将包含具有相同签名的分发和扩展接收者方法（方法一定义于 Button 类中，方法二定义于 MainActivity 类中），如下所示：

```
class MainActivity: Activity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.main activity)
        val buttonView = findViewById(R.id.button view) as Button
        buttonView.setUp()
    }

    private fun Button.setUp() {
        setText("Click me!")
        setOnClickListener { setText("Hello") } // 1
    }
    private fun setText(text: String) {
        textView.setText(text)
    }
}
```

针对注释 1，setText 定义为分发接收者和扩展接收者中的方法，这里的问题是，应调用哪一个方法？

最终结果为：setText 函数将被扩展接收者调用；相应地，按钮单击行为将改变单击按钮的文本，其原因在于：扩展接收者优于分发接收者。尽管如此，在该情形下，仍可通过限定语法（即包含标记的 this 关键字，用以区分所引用的接收者）使用分发接收者，如下所示：

```
private fun Button.setUp() {
    setText("Click me!")
    setOnClickListener {
        this@MainActivity.setText("Hello")
    }
}
```

通过这种方式，可有效地解决分发和扩展接收者之间的区分问题。

7.3.2　成员扩展函数和属性的底层机制

成员扩展函数和属性采用与顶级扩展函数和属性相同的编译方式，唯一的差别在于位于某个类中，且为非 static 状态。下列代码显示了扩展函数的简单示例。

```
class A {
    fun boo() {}

    fun Int.foo() {
        boo()
     }
}
```

这将编译为（经适当简化）：

```
public final class A {
    public final void boo() {
        ...
    }

    public final void foo(int $receiver) {
        this.boo();
    }
}
```

注意，虽然它们只是作为第一个参数的接收者的方法，但是我们可以用其他函数对其进行处理。

7.4 泛型扩展函数

当编写工具函数时，通常需要使用到泛型这一概念。其中，较为常见的例子是集合扩展，例如 List、Map 和 Set。下列代码显示了 List 的扩展属性。

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

上述代码针对泛型定义了扩展属性，此类扩展适用于大多数不同场合。作为例子，启动另一个 Activity 可视作一项重复任务，此类任务往往会在项目中多处加以实现。对于 Activity 的启用操作，Android IDE 所提供的方法并未简化事物。下列代码用于启动新的 Activity，称之为 SettingsActivity。

```
startActivity(Intent (this, SettingsActivity::class.java))
```

注意，这一简单且重复性的任务涵盖了大量代码，且清晰性较差。但可创建 Intent 的扩展函数，不包含参数的 Activity 将以更简单的方式启动，同时使用了基于 reified 类型的泛型内联扩展函数，如下所示：

```
inline fun <reified T : Any> Context.getIntent()
   = Intent(this, T::class.java)

inline fun <reified T : Any> Context.startActivity()
   = startActivity(getIntent<T>())
```

随后，可通过下列代码并以更加简单的方式启用 Activity。

```
startActivity<SettingsActivity>()
```

或者，也可通过下列方式创建 intent。

```
val intent = getIntent<SettingsActivity>()
```

据此，可以最小的代价制定此类常见任务。进一步讲，诸如 Anko()这一类库提供了扩展函数，同时提供了一种简单方式启用包含附加参数或标识的 Activity，如下所示：

```
startActivity<SettingsActivity>(userKey to user)
```

该库的内部实现超出了本书的讨论范围，但可通过向当前项目添加 Anko 库（参见 https://github.com/MarcinMoskala/ActivityStarter）依赖关系，进而简单地使用此类扩展。该示例的重点在于，几乎所有的重复性代码均可通过扩展被简单的代码所替代。除此之外，

还存在一种备选方案可启用 Activity，例如 ActivityStarter 库（参见 https://github.com/MarcinMoskala/ActivityStarter），该库基于参数注入同时支持 Kotlin，如下所示：

```
class StudentDataActivity : BaseActivity() {
    lateinit @Arg var student: Student
    @Arg(optional = true) var lesson: Lesson = Lesson.default()
}
```

或者作为替代方案，还支持 Kotlin 属性托管中的延迟注入（参见第 8 章），如下所示：

```
class StudentDataActivity : BaseActivity() {
    @get:Arg val student: Student by argExtra()
    @get:Arg(optional = true)
    var lesson: Lesson by argExtra(Lesson.default())
}
```

包含此类参数的 Activity 可通过生成后的静态参数予以启用，如下所示：

```
StudentDataActivityStarter.start(context, student, lesson)
StudentDataActivityStarter.start(context, student)
```

下面考察另一个示例。在 Android 中，常常需要以 JSON 格式存储对象，例如将其发送至 API 中，或者存储于某个文件中。对于 JSON 中的序列化和反序列化，较为常用的库是 Gson。下面讨论 Gson 库的标准应用方式，如下所示：

```
val user = User("Marcin", "Moskala")
val json: String = globalGson.toJson(user)
val userFromJson = globalGson.fromJson(json, User::class.java)
```

根据包含 inline 修饰符的扩展函数，可在 Kotlin 中对此加以改进。下列扩展函数示例使用了 GSON，将对象打包/解包至 JSON 格式中的 String，如下所示：

```
inline fun Any.toJson() = globalGson.toJson(this)!!

inline fun <reified T : Any> String.fromJson()
  = globalGson.fromJson(this, T::class.java)

// Usage
val user = User("Marcin", "Moskala")
val json: String = user.toJson()
val userFromJson: User = json.fromJson<User>()
```

其中，globalGson 实例表示为全局 Gson 实例。这可视作一类常见操作，但通常会定义某些序列化器和反序列化器，这可视作一种简单、高效的方式，从而一次性地对其进行定义并构建 Gson 实例。

一些示例向程序员展示了泛型扩展函数的各种可能性，它们更像是代码所提取的下一个级别，其中包括：

- 表示为顶级函数，但也可在某个对象上加以调用，因而便于管理。
- 表示为泛型函数，因而应用范围较广。
- 当采用 inline 时，可定义 reified 类型参数。

这也体现了泛型扩展函数在 Kotlin 中普遍使用的原因。另外，标准库也提供了大量的泛型扩展函数，稍后将会讨论某些集合扩展函数。这一部分内容较为重要，不仅是因为可提供与泛型扩展函数使用方面的知识，同时还描述了 Kotlin 中列表处理方式以及应用方式。

集合处理在程序设计中是一类常见任务，开发人员往往会首先学习到集合的遍历方式，进而对相关数据元素进行操作。例如，列表中的所有用户的输出可能会用到 for 循环，如下所示：

```
for (user in users) {
    println(user)
}
```

如果输出通过学校考试的用户，则一般会在循环中加入下列 if 条件语句：

```
for (user in users) {
    if ( user.passing ) {
        println(user)
    }
}
```

这仍可视作正确的实现方式，但当任务变得复杂时，问题也会随之出现。例如，输出 3 名成绩最优的学生。在循环中，实现过程变得复杂起来。对此，Kotlin 中一种简单的实现方法是采用流处理。下面对此加以考察，学生列表如下所示：

```
data class Student(
    val name: String,
    val grade: Double,
    val passing: Boolean
)

val students = listOf(
    Student("John", 4.2, true),
    Student("Bill", 3.5, true),
    Student("John", 3.2, false),
    Student("Aron", 4.3, true),
    Student("Jimmy", 3.1, true)
)
```

下面利用 Java 中的命令式方案对学生进行过滤操作（使用循环和排序方法），如下所示：

```
val filteredList = ArrayList<Student>()
for (student in students) {
    if(student.passing) filteredList += student
}

Collections.sort(filteredList) { p1, p2 ->
    if(p1.grade > p2.grade) -1 else 1
}

for (i in 0..2) {
    val student = filteredList[i]
    println(student)
}

// Prints:
// Student(name=Aron, grade=4.3, passing=true)
// Student(name=John, grade=4.2, passing=true)
// Student(name=Bill, grade=3.5, passing=true)
```

采用 Kotlin 流处理，可通过更加简单的方式实现相同结果，如下所示：

```
students.filter { it.passing } // 1
   .sortedByDescending { it.grade } // 2
   .take(3) // 3
   .forEach(::println) // 4
```

针对注释 1，获取通过考试的学生名单；对于注释 2，根据其成绩对学生排序；对于注释 3，仅取学生中的前 3 名；对于注释 4，输出结果。

这里，核心内容包括：各个流处理函数可获取较小的功能项，例如 sortedByDescending、take、forEach，经适当整合后，可发挥强大的功能。与经典的循环应用相比，最终结果更加简单且具有可读性。

实际上，流处理是一种常见的语言特性，包括 C#、JavaScript、Scala 以及 Java（自版本 8 后）等。一些流行的响应式程序库也采用这一概念处理数据，例如 RxJava。下面将深入讨论 Kotlin 中的集合处理。

7.4.1　Kotlin 集合类型层次结构

Kotlin 类型层次结构实现了良好的设计，标准集合实际上是源自本地语言（例如 Java）中的集合，并隐藏于接口之后。其创建过程由标准的顶级函数完成（例如 listOf、setOf、mutableListOf 等），因而可在公共模块中创建和使用（模块可编译至多个平台上）。另外，Kotlin 接口可像 Java 中的对等接口那样工作（例如 List、Set 等），这也使得 Kotlin 集合更

加高效并兼容于外部库。同时，Kotlin 集合接口层次结构也可用于公共模块中。对应的层次结构较为简单，读者应对此加以理解，如图 7.6 所示。

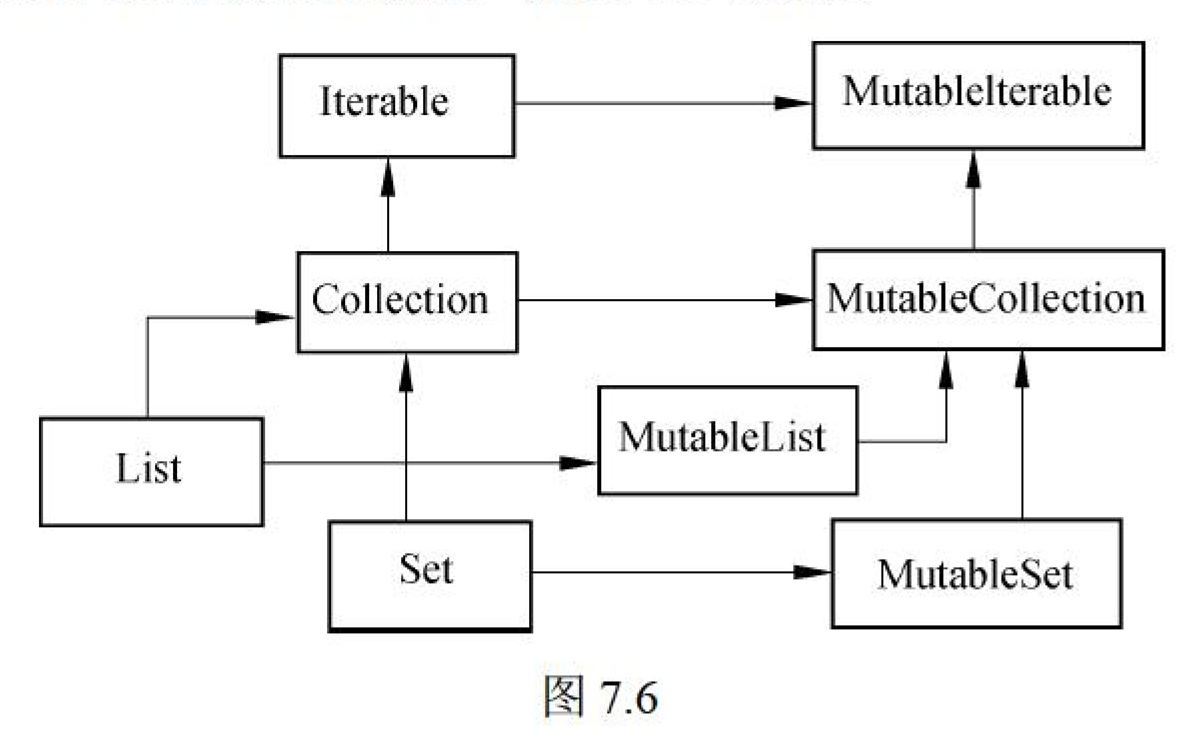


图 7.6

其中，较为通用的接口是 Iterable，表示为可迭代的数据元素序列，实现了 Iterable 的任意对象均可用于 for 循环中，如下所示：

```
for (i in iterable) { /* ... */ }
```

多种不同的类型均实现了 Iterable 接口，包括全部集合、数列（1...10，'a'...'z'等），甚至是 String，进而可遍历其中的元素，如下所示：

```
for (char in "Text") { print("($char)") } // Prints: (T)(e)(x)(t)
```

Collection 接口表示为数据元素集合，并扩展了 Iterable，其中加入了 size 属性，以及 contains、containsAll 和 isEmpty 方法。

List 和 Set 则是继承自 Collection 的两个主要接口，二者间的差别在于，Set 呈无序状态，且不包含重复元素（根据 equals 方法）。另外，List 和 Set 接口不包含修改对象状态的方法。这也是默认状态 Kotlin 集合视为不可变的原因。对于 List 接口，Android 中较为常见的是 ArrayList。ArrayList 是一类可变集合，但隐藏于接口 List 之后，其行为实际上处于不可变状态——并未展示任何可实施变化的相关方法（除非对其进行向下转型）。

在 Java 中，集合呈现为可变状态，但 Kotlin 接口默认时仅提供了不可变行为（而非修改集合状态的方法，例如 add 和 removeAt 方法），如下所示：

```
val list = listOf('a', 'b', 'c')
println(list[0]) // Prints: a
println(list.size) // Prints: 3
list.add('d') // Error
list.removeAt(0) // Error
```

全部不可变接口（例如 Collection、List 等）均包含对应的可变接口（例如 Mutable Collection、MutableList 等），且均继承自对应的可变接口。这里，可变意味着可对实际对

象进行调整。下列内容展示了标准库中可变集合的接口：

- MutableIterable 支持包含变化内容的迭代操作。
- MutableCollection 涵盖了添加和移除数据元素的方法。
- MutableList 和 MutableSet 对应于 List 和 Set 的可变形式。

据此，可通过 add 和 remove 方法修改集合，进而修正上述示例，如下所示：

```
val list = mutableListOf('a', 'b', 'c')
println(list[0]) // Prints: a
println(list.size) // Prints: 3
list.add('d')
println(list) // Prints: [a, b, c, d]
list.removeAt(0)
println(list) // Prints: [b, c, d]
```

不可变和可变接口仅提供了少量方法，但 Kotlin 标准库对此提供了大量的有效扩展，如图 7.7 所示。

```
users.
reduceRight(operation: (User, User) -> User) for List<T> in kot..  User
reduce(operation: (User, User) -> User) for Iterable<T> in kotl..  User
reduce { acc, User -> ... } (operation: (User, User) -> User) f..  User
reduceRight { User, acc -> ... } (operation: (User, User) -> Us..  User
reduceRightIndexed(operation: (Int, User, User) -> User) for Li..  User
reduceRightIndexed { index, User, acc -> ... } (operation: (Int..  User
single() for List<T> in kotlin.collections  User
singleOrNull() for List<T> in kotlin.collections  User?
single {...} (predicate: (User) -> Boolean) for Iterable<T> in ..  User
slice(indices: IntRange) for List<T> in kotlin.collections  List<User>
slice(indices: Iterable<Int>) for List<T> in kotlin.colle..  List<User>
takeLast(n: Int) for List<T> in kotlin.collections  List<User>
take(n: Int) for Iterable<T> in kotlin.collections  List<User>
takeLastWhile {...} (predicate: (User) -> Boolean) for Li..  List<User>
sortedBy {...} (crossinline selector: (User) -> R?) for I..  List<User>
filter {...} (predicate: (User) -> Boolean) for Iterable<..  List<User>
indices for Collection<*> in kotlin.collections  IntRange
times(other: Iterable<L>) for Iterable<T> in swe..  List<Pair<User, L>>
powerset() for Collection<T> in sweet  Set<Set<User>>
all {...} (predicate: (User) -> Boolean) for Iterable<T> in ..  Boolean
any() for Iterable<T> in kotlin.collections  Boolean
any {...} (predicate: (User) -> Boolean) for Iterable<T> in ..  Boolean
asIterable() for Iterable<T> in kotlin.collections  Iterable<User>
asSequence() for Iterable<T> in kotlin.collections  Sequence<User>
associate {...} (transform: (User) -> Pair<K, V>) for Iter..  Map<K, V>
associateBy {...} (keySelector: (User) -> K) for Iterab..  Map<K, User>
Ctrl+Down and Ctrl+Up will move caret down and up in the editor >>
```

图 7.7

与 Java 相比，这大大简化了集合处理任务。

Kotlin 通过扩展实现了集合处理方法，该方案包含诸多优点。例如，如果需要实现一个自定义集合（例如 List），仅须实现包含少量方法的 Iterable 接口即可。同时，仍然可使用提供于 Iterable 接口的全部扩展。

另一个原因是，当定义为接口的扩展时，这些函数可采取更加灵活的方式使用。例如，大多数集合处理函数实际上表示为基于 Iterable 的扩展，并通过除 Collection 之外的多种类型予以实现，例如 String 或 Range。因此，可针对 Iterable 以及 IntRange 使用所有扩展函数，对应示例如下所示：

```
(1..5).map { it * 2 }.forEach(::print) // Prints: 246810
```

这也极大地提升了此类扩展的应用范围。但下列事实也体现了一定的缺点：集合流处理方法实现为扩展函数。尽管扩展可采用静态方式加以处理，但针对某一特定类型覆写某个扩展函数则是错误的，其原因在于，若位于某个接口之后并直接予以访问时，其行为将有所不同。

下面对用于集合处理的某些扩展函数加以分析。

7.4.2　map、filter 和 flatMap 函数

前述内容曾简要地介绍了 map、filter 以及 flatMap，均是较为基础的流处理函数。其中，map 函数返回一个元素列表，该列表数据根据参数中的函数进行适当调整，如下所示：

```
val list = listOf(1,2,3).map { it * 2 }
println(list) // Prints: [2, 4, 6]
```

filter 函数仅支持与所提供的断言条件相匹配的元素，如下所示：

```
val list = listOf(1,2,3,4,5).map { it > 2 }
println(list) // Prints: [3, 4, 5]
```

flatMap 函数返回一个经由转换函数生成的、包含全部元素的单一列表，并在原始集合的各个元素上被调用，如下所示：

```
val list = listOf(10, 20).flatMap { listOf(it, it+1, it + 2) }
println(list) // Prints: [10, 11, 12, 20, 21, 22]
```

该函数常用于实现几何列表的“扁平化”，如下所示：

```
shops.flatMap { it.products }
schools.flatMap { it.students }
```

下面考察此类扩展函数的简化实现，如下所示：

```
inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> { //1
    val destination = ArrayList<R>()
    for (item in this) destination.add(transform(item)) // 2
    return destination
}

inline fun <T> Iterable<T>.filter(predicate: (T) -> Boolean): List<T> { //
1
    val destination = ArrayList<T>()
    for (item in this) if(predicate(item)) destination.add(item) // 2
    return destination
}

inline fun <T, R> Iterable<T>.flatMap(transform: (T) -> Collection<R>):
List<R> {
// 1
    val destination = ArrayList<R>()
    for (item in this) destination.addAll(transform(item)) // 2
    return destination
}
```

针对注释 1，该类所有函数均为 inline；针对注释 2，所有函数均在循环内部使用，并返回包含适当元素的新列表。

大多数包含函数类型的 Kotlin 标准库扩展函数均为 inline，进而可高效地使用 Lambda 扩展。最终，全部集合流处理一般在运行期编译为嵌套循环。下列示例展示了简单的处理过程。

```
students.filter { it.passing }
   .map { "${it.name} ${it.surname}" }
```

在编译/反编译至 Java 后，对应代码如下所示：

```
Collection destination1 = new ArrayList();
Iterator it = students.iterator();
while(it.hasNext()) {
    Student student = (Student) it.next();
    if(student.getPassing()) {
        destination1.add(student);
    }
}
Collection destination2 = new ArrayList(destination1.size());
it = destination2.iterator();
while(it.hasNext()) {
```

```
    Student student = (Student) it.next();
    String var = student.getName() + " " + student.getSurname();
    destination2.add(var);
}
```

7.4.3　forEach 和 onEach 函数

前述章节已对 forEach 函数有所讨论，并视作 for 循环的一种替代方案，因而可在各个列表元素上执行某项操作，如下所示：

```
listOf("A", "B", "C").forEach { print(it) } // prints: ABC
```

自 Kotlin 1.1 以来，存在一个类似的函数 onEach，也可在每个元素上调用某项操作，并返回扩展接收者（当前列表）。因此，可在流处理中在每个元素上调用操作。日志功能则是一类常见示例，如下所示：

```
 (1..10).filter { it % 3 == 0 }
   .onEach(::print) // Prints: 369
   .map { it / 3 }
   .forEach(::print) // Prints: 123
```

7.4.4　withIndex 以及索引变化版本

某些时候，数据元素的处理方式取决于列表上的索引。对此，一种处理该问题的常见方法是使用 withIndex 函数，该函数返回包含索引的值列表，如下所示：

```
listOf(9,8,7,6).withIndex() // 1
   .filter { (i,  ) -> i % 2 == 0 } // 2
   .forEach { (i, v) -> print("$v at $i,") }
// Prints: 9 at 0, 7 at 2,
```

针对注释 1，withIndex 函数将各个数据元素打包至 IndexedValue 中，其中包含了当前元素及其索引；针对注释 2，在 Lambda 中，IndexedValue 析构为索引和数值，尽管该值尚未使用，但仍添加了一个下划线并可予以忽略，且这种代码编写方式具有较好的可读性。该行代码仅过滤掉包含偶数索引的数据元素。

除此之外，还存在不同流处理方法的变化版本，并可提供一个索引，如下所示：

```
val list1 = listOf(2, 2, 3, 3)
    .filterIndexed { index,    -> index % 2 == 0 }
println(list1) // Prints: [2, 3]

val list2 = listOf(10, 10, 10)
```

```
    .mapIndexed { index, i -> index * i }
println(list2) // Prints: [0, 10, 20]

val list3 = listOf(1, 4, 9)
    .forEachIndexed { index, i -> print("$index: $i,") }
println(list3) // Prints: 0: 1, 1: 4, 2: 9
```

7.4.5 sum、count、min、max 和排序函数

sum 函数计算列表中全部元素之和，并可在 List<Int>，List<Long>，List<Short>，List<Double>，List<Float>，List<Byte>上被调用，如下所示：

```
val sum = listOf(1,2,3,4).sum()
println(sum) // Prints: 10
```

通常，还需要对元素的某些属性求和，例如所有用户的分数求和。对此，可将用户列表映射为数值列表，并于随后求和，如下所示：

```
class User(val points: Int)
val users = listOf(User(10), User(1 000), User(10 000))

val points = users.map { it.points }. sum()
println(points) // Prints: 11010
```

通过调用 map 函数，无须生成中间集合，因而更加高效并可直接对分数求和。对此，可使用包含相应的选择器的 sumBy，如下所示：

```
val points = users.sumBy { it.points }
println(points) // Prints: 11010
```

其中，sumBy 期望从选择器中返回 Int，并返回包含数据元素求和结果的 Int。如果最终结果为 Double 而非 Int，则可使用 sumByDouble 并返回 Double，如下所示：

```
class User(val points: Double)
val users = listOf(User(10.0), User(1 000.0), User(10 000.0))

val points = users.sumByDouble { it.points }
println(points) // Prints: 11010.0
```

count 函数也提供了类似的功能，并在需要计算与断言条件匹配的元素时使用，如下所示：

```
val evens = (1..5).count { it % 2 == 1 }
val odds = (1..5).count { it % 2 == 0 }
```

```
println(evens) // Prints: 3
println(odds) // Prints: 2
```

不包含断言条件的 count 函数则返回集合或迭代的尺寸，如下所示：

```
val nums = (1..4).count()
println(nums) // Prints: 4
```

下一个较为重要的函数是 min 和 max，并返回列表中的最小和最大值。这一类函数常用于包含自然排序的元素列表（实现了 Comparable<T>接口），如下所示：

```
val list = listOf(4, 2, 5, 1)
println(list.min()) // Prints: 1
println(list.max()) // Prints: 5
println(listOf("kok", "ada", "bal", "mal").min()) // Prints: ada
```

类似地，函数 sorted 返回排序列表，但需要在实现了 Comparable<T>接口的元素集合上加以调用。下列代码展示了如何获取字母排序的字符串列表。

```
val strs = listOf("kok", "ada", "bal", "mal").sorted()
println(strs) // Prints: [ada, bal, kok, mal]
```

若数据项未经比较，情况又当如何？此处，存在两种方法对其进行排序。方法一根据比较数据项排序。前述示例根据成绩对学生进行排序，如下所示：

```
students.filter { it.passing }
   .sortedByDescending { it.grade }
   .take(3)
   .forEach(::println)
```

在该示例中，可比较 grade 属性对学生进行排序，其中使用了 sortedByDescending，其工作方式类似于 sortedBy。唯一差别在于，排序以降序操作（从大到小）。函数内的选择器可返回与自身比较的任意值。在下列示例中，则通过 String 指定顺序。

```
val list = listOf(14, 31, 2)
print(list.sortedBy { "$it" }) // Prints: [14, 2, 31]
```

类似的函数也可根据选择器获取最小和最大数据元素，如下所示：

```
val minByLen = listOf("ppp", "z", "as")
    .minBy { it.length }
println(minByLen) // Prints: "z"

val maxByLen = listOf("ppp", "z", "as")
    .maxBy { it.length }
println(maxByLen) // Prints: "ppp"
```

指定排序的第二种方式是定义 Comparator，进而确定元素的比较方式。另外，接收比较器的函数变化版本则应包含 With 后缀。相应地，比较器可通过适配器函数加以定义，进而将 Lambda 转换为 SAM 类型，如下所示：

```
val comparator = Comparator<String> { e1, e2 ->
    e2.length - e1.length
}
val minByLen = listOf("ppp", "z", "as")
    .sortedWith(comparator)
println(minByLen) // Prints: [ppp, as, z]
```

Kotlin 还引入了标准库顶级函数（例如 compareBy 和 compareByDescending），从而简化 Comparator 的创建过程。下列代码显示了如何创建比较器，并通过 surname 和 name 对学生进行字母排序。

```
data class User(val name: String, val surname: String) {
    override fun toString() = "$name $surname"
}
val users = listOf(
   User("A", "A"),
   User("B", "A"),
   User("B", "B"),
   User("A", "B")
)

val sortedUsers = users
   .sortedWith(compareBy({ it.surname }, { it.name }))

print(sortedUsers) // [A A, B A, A B, B B]
```

注意，还可使用属性引用，而非 Lambda 表达式，如下所示：

```
val sortedUsers = users
   .sortedWith(compareBy(User::surname, User::name))
print(sortedUsers) // [A A, B A, A B, B B]
```

groupBy 则是另一种较为重要的函数，并根据当前选择器对数据元素进行分组。groupBy 将返回 Map，并从所选的键映射至元素列表，该列表经选择后将映射至下列键：

```
val grouped = listOf("ala", "alan", "mulan", "malan")
    .groupBy { it.first() }
println(grouped) // Prints: {'a': ["ala", "alan"], "m": ["mulan", "malan"]}
```

下面考察一个相对复杂的示例，即获取每个班中最优学生列表。下列代码展示了如何从学生列表中获取最终结果。

```
class Student(val name: String, val classCode: String, val meanGrade: Float)

val students = listOf(
    Student("Homer", "1", 1.1F),
    Student("Carl", "2", 1.5F),
    Student("Donald", "2", 3.5F),
    Student("Alex", "3", 4.5F),
    Student("Marcin", "3", 5.0F),
    Student("Max", "1", 3.2F)
)

val bestInClass = students
    .groupBy { it.classCode }
    .map { ( , students) -> students.maxBy { it.meanGrade }!! }
    .map { it.name }

print(bestInClass) // Prints: [Max, Donald, Marcin]
```

7.4.6　其他流处理函数

除此之外，还存在其他不同的流处理函数，Kotlin 在其网站中提供了大量的说明文档。其中，大多数扩展函数均具有自解释特性，开发人员可直接据此猜测其含义。在 Android Studio 中，可按 Ctrl 键（Mac 中则为 Command 键）并单击目标函数，以查看其对应实现。

当在可变集合上进行操作时，将会看到集合处理间的差异，其原因在于：此处使用了针对可变类型（例如 MutableIterable 和 MutableCollection）所定义的附加扩展。其中，较为重要的区别在于，修改对象的函数以命令式形式构建（例如 sort），而返回包含变化值新集合的函数则常以动词过去式定义（例如 sorted）。相关示例如下所示。

- sort：该函数对可变对象进行排序，并返回 Unit。
- sorted：该函数返回一个有序集合，并且不会改变其上调用的集合。

```
val list = mutableListOf(3,2,4,1)
val list2 = list.sorted()
println(list) // [3,2,4,1]
println(list2) // [1,2,3,4]
list.sort()
println(list) // [1,2,3,4]
```

7.4.7　集合流处理示例

之前曾讨论了一些流处理函数，但对于复杂示例，则会涉及一些操作技巧。下面讨论某些较为复杂的流处理示例。

假设需要根据成绩获取前三名的学生，此处关键差异在于，学生最终的顺序必须和开始时一样——需要注意的是，在以成绩进行操作时，该顺序将丢失。若保留对应顺序，则须持有数值和索引。其实现过程如下所示：

```
data class Student(
   val name: String,
   val grade: Double,
   val passing: Boolean
)

val students = listOf(
   Student("John", 4.2, true),
   Student("Bill", 3.5, true),
   Student("John", 3.2, false),
   Student("Aron", 4.3, true),
   Student("Jimmy", 3.1, true)
)

val bestStudents = students.filter { it.passing } // 1
   .withIndex() // 2
   .sortedBy { it.value.grade } // 3
   .take(3) // 4
   .sortedBy { it.index } // 5
   .map { it.value } // 6

// Print list of names
println(bestStudents.map { it.name }) // [John, Bill, Jimmy]
```

针对注释 1，过滤并获取通过考试的学生；对于注释 2，向元素中加入索引，以重现数据元素顺序；对于注释 3，根据成绩对学生进行排序；对于注释 4，仅获取前 10 名的学生；对于注释 5，根据索引排序并重现排序；对于注释 6，将包含索引的数值映射至当前值。

注意，上述实现较为简洁，且集合上执行的各项操作具有良好的可读性。

集合流处理最大的优势在于，可方便地管理该处理的复杂度。如前所述，大多数操作的复杂度为 O(n)，例如 map 或 filter。而排序的复杂度为 O(n*log(n))。另外，流处理操作的复杂度则表示为各个步骤中的最大复杂度，因此，上述处理的复杂度为 O(n*log(n))——sortedBy 步骤中包含了最大复杂度。

假设某个列表包含了不同分类的多名选手，如下所示：

```
class Result(
   val player: Player,
   val category: Category,
   val result: Double
)
class Player(val name: String)
enum class Category { SWIMMING, RUNNING, CYCLING }
```

对应的示例数据如下所示：

```
val results = listOf(
   Result("Alex", Category.SWIMMING, 23.4),
   Result("Alex", Category.RUNNING, 43.2),
   Result("Alex", Category.CYCLING, 15.3),
   Result("Max", Category.SWIMMING, 17.3),
   Result("Max", Category.RUNNING, 33.3),
   Result("Bob", Category.SWIMMING, 29.9),
   Result("Bob", Category.CYCLING, 18.0)
)
```

下列代码显示了如何在各个分类中获取最优选手。

```
val bestInCategory = results.groupBy { it.category } // 1
   .mapValues { it.value.maxBy { it.result }?.player } // 2
print(bestInCategory)
// Prints: {SWIMMING=Bob, RUNNING=Alex, CYCLING=Bob}
```

在注释 1 中，将最终结果分组至不同分类中，返回类型为<Category>和 List<Result>；在注释 2 中，将映射 map 函数值。其中，将得到当前分类中的最优结果，进而获取与该结果所关联的选手。mapValues 函数的返回结果为 Map<Category, Player?>。

基于集合处理函数，上述示例展示了如何在 Kotlin 中通过简单方式处理与集合相关的复杂问题。在与 Kotlin 工作一段时间后，相信程序员会对大多数函数有所了解，进而发现集合处理问题将变得十分简单。当然。上述较为复杂的函数一般较为少见；在日常编程中，更为常见的则是相对简单且仅包含几个步骤的处理过程。

7.4.8　序列

Sequence 表示为一类接口，并可用于引用数据元素集合，同时也是 Iterable 的替代方案。对于 Sequence 而言，存在大多数集合处理函数的独立实现（例如，map、filterMap、filter、sorted 等）。关键的差别在于这一类函数的构造方式，即返回序列，并在前一序列的

基础上打包序列。据此，可得到以下事实：

- 序列尺寸无须事先知晓。
- 序列处理将更加高效，尤其是对于大型集合，其中需要执行多次转换（稍后将对此加以讨论）。

在 Android 中，序列常用于处理大型集合，或者处理尺寸事先未知的元素（例如读取大型文档中的多行数据）。序列的创建方式多种多样，其中，较为简单的方法是在 Iterable 上调用 asSequence 函数；或者使用顶级函数 sequenceOf 并采用类似于列表的方式生成序列。

序列的大小无须事先知晓，对应值仅在必要时加以计算，如下所示：

```
val = generateSequence(1) { it + 1 } // 1. Instance of GeneratorSequence
   .map { it * 2 } // 2. Instance of TransformingSequence
   .take(10) // 3. Instance of TakeSequence
   .toList() // 4. Instance of List

println(numbers) // Prints: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

针对注释 1，generateSequence 可视为一种序列生成方式，该序列包含了从 1 至无穷大的下一个数字；针对注释 2，map 函数将序列打包至另一个序列中，该序列从第一个序列中获取数值，并在转换后计算当前值；针对注释 3，函数 take(10)将序列打包至另一个序列，该序列在第 10 个元素处结束。如果缺失该行执行代码，处理时间将不确定，进而在某个不确定的序列上进行操作；针对注释 4，函数 toList 负责处理各个数值，并返回至最终的列表中。

需要着重强调的是，数据元素在最后一个步骤（终端操作）中逐一加以处理。下面查看另一个示例，其中，各项操作将针对日志功能输出数值。该示例始于下列代码：

```
val seq=generateSequence(1) { println("Generated ${it+1}"); it + 1 }
   .filter { println("Processing of filter: $it"); it % 2 == 1 }
   .map { println("Processing map: $it"); it * 2 }
   .take(2)
```

在控制台中，输出结果又当如何？此时将不会输出任何内容——相关数值并未被计算。其原因在于，全部中间操作将被延迟。为了获取对应结果，须使用诸如 toList 这一类终端操作，如下所示：

```
seq.toList()
```

随后，控制台中将显示下列内容：

```
Processing of filter: 1
Processing map: 1
Generated 2
```

```
Processing of filter: 2
Generated 3
Processing of filter: 3
Processing map: 3
```

注意，数据元素将被逐一处理。在标准的列表处理中，操作顺序将截然不同，如下所示：

```
(1..4).onEach { println("Generated $it") }
   .filter { println("Processing filter: $it"); it % 2 == 1 }
   .map { println("Processing map: $it"); it * 2 }
```

上述代码将输出下列结果：

```
Generated 1
Generated 2
Generated 3
Generated 4
Processing filter: 1
Processing filter: 2
Processing filter: 3
Processing filter: 4
Processing map: 1
Processing map: 3
```

与经典的集合处理相比，这也解释了序列更加高效的原因——无须在中间步骤中生成集合。数值将采用即时方式逐一加以处理。

7.5　包含接收者的函数字面值

类似于包含函数类型的函数，并可将其视作一个对象，扩展函数也涵盖了其类型，并可通过这一方式予以保存。这称作包含接收者的函数类型，且类似于简单函数类型，但接收者类型位于参数之前（与扩展定义类似），如下所示：

```
var power: Int.(Int) -> Int
```

基于接收者的函数类型使得函数和类型具有完整的衔接性——全部函数均可表示为对象，同时可通过包含接收者的 Lambda 表达式或者包含接收者的匿名函数加以定义。

在包含接收者定义的 Lambda 表达式中，唯一的差别在于，可通过 this 引用接收者，并可显式地使用接收者元素。对于 Lambda 表达式，类型须在某个参数中指定，因为不存在对应语法可确定接收者类型。在下列代码中，power 定义为包含接收者的 Lambda 表达式。

```
power = { n -> (1..n).fold(1) { acc, _ -> this * acc } }
```

匿名函数也可定义接收者，其类型置于函数名之前。在此类函数中，可在函数体中使用 this，以引用扩展接收者对象。需要注意的是，匿名扩展函数指定了当前接收者类型，因而可推断属性类型。在下列代码中，power 定义为匿名扩展函数。

```
power = fun Int.(n: Int) = (1..n).fold(1) { acc, _ -> this * acc }
```

从表面上看，基于接收者的函数类型可用作接收者类型的一个方法，如下所示：

```
val result = 10.power(3)
println(result) // Prints: 1000
```

函数类型常用作函数参数。在下列示例中，采用了参数函数，并在其创建后配置某个数据元素，如下所示：

```
fun ViewGroup.addTextView(configure: TextView.()->Unit) {
    val view = TextView(context)
    view.configure()
    addView(view)
}

// Usage
val linearLayout = findViewById(R.id.contentPanel) as LinearLayout

linearLayout.addTextView { // 1
    text = "Marcin" // 2
    textSize = 12F // 2
}
```

针对注释 1，此处采用了 Lambda 表达式作为参数；针对注释 2，在 Lambda 表达式中，可直接调用接收者方法。

7.5.1　Kotlin 标准库函数

Kotlin 标准库提供了一组包含泛型非限定接收者的扩展函数（例如 let、apply、also、with、run 和 to，且泛型不包含限定条件），并可视为一类小型、简单的扩展。在 Kotlin 项目中，这一类扩展十分有用。第 2 章曾简要介绍了 let 函数，并用作空检测的替代方案，如下所示：

```
savedInstanceState?.let{ state ->
    println(state.getBoolean("isLocked"))
}
```

let 的全部任务即是调用特定函数，并返回对应值。在前述示例中，let 与安全调用操作符结合使用，仅当属性 savedInstanceState 非空时方被调用。这里，let 函数实际上仅是包含参数函数的泛型扩展函数，如下所示：

```
inline fun <T, R> T.let(block: (T) -> R): R = block(this)
```

在 stdlib 中，存在多个与 let 类似的函数，其中包括 applay、also、with 和 run。鉴于函数间的相似性，此处可综合对其进行描述，相关定义如下所示：

```
inline fun <T> T.apply(block: T.() -> Unit): T {
    block();
    return this
}
inline fun <T> T.also(block: (T) -> Unit): T {
    block(this);
    return this
}
inline fun <T, R> T.run(block: T.() -> R): R = block()
inline fun <T, R> with(receiver: T, block: T.() -> R): R = receiver.block()
```

相应地，应用示例如下所示：

```
val mutableList = mutableListOf(1)
val mutableList = mutableListOf(1)
val letResult = mutableList.let {
    it.add(2)
    listOf("A", "B", "C")
}
println(letResult) // Prints: [A, B, C]
val applyResult = mutableList.apply {
    add(3)
    listOf("A", "B", "C")
}
println(applyResult) // Prints: [1, 2, 3]
val alsoResult = mutableList.also {
   it.add(4)
   listOf("A", "B", "C")
}
println(alsoResult) // Prints: [1, 2, 3, 4]
val runResult = mutableList.run {
   add(5)
   listOf("A", "B", "C")
}
println(runResult) // Prints: [A, B, C]
```

```
val withResult = with(mutableList) {
   add(6)
   listOf("A", "B", "C")
}
println(withResult) // Prints: [A, B, C]
println(mutableList) // Prints: [1, 2, 3, 4, 5, 6]
```

表 7.1 显示了其中的差别。

表 7.1

返回对象/参数函数类型	包含接收者的函数字面值（接收者对象表示为 this）	函数字面值（接收者对象表示为 it）
接收者对象	apply	also
函数字面值的结果	run/with	let

这一类函数较为相似，大多数时候可交互使用。根据规则，对某些特殊用例须优先使用某些函数。

1. let 函数

当需要使用标准函数，并在流处理中作为扩展函数时，推荐使用 let 函数，如下所示：

```
val newNumber = number.plus(2.0)
    .let { pow(it, 2.0) }
    .times(2)
```

类似于其他扩展，let 函数常与某种安全调用操作符结合使用，如下所示：

```
val newNumber = number?.plus(2.0)
   ?.let { pow(it, 2.0) }
```

除此之外，当仅需要解包可空读-写属性时，推荐使用 let 函数。此时，无法对该属性进行智能转换，并须执行下列屏蔽（shadowing）操作：

```
var name: String? = null

fun Context.toastName() {
    val name = name
    if(name != null) {
        toast(name)
    }
}
```

其中，name 变量表示为屏蔽属性名，其必要性体现在当名称为读写属性时——智能转换仅可用在可变或局部变量上。

据此，可利用 let 和安全调用操作符替换上述代码，如下所示：

```
name?.let { setNewName(it) }
```

注意，当使用 Elvis 操作符时，可简单地添加 return，否则在 name 为 null 时将会抛出异常，如下所示：

```
name?.let { setNewName(it) } ?: throw Error("No name setten")
```

类似地，let 函数还可替换下列语句：

```
val comment = if(field == null) getComment(field) else "No comment
```

基于 let 函数的实现如下所示：

```
val comment = field?.let { getComment(it) } ?: "No comment"
```

在转换接收者的方法链中，推荐使用基于该方式的 let 函数，如下所示：

```
val text = "hello {name}"

fun correctStyle(text: String) = text
   .replace("hello", "hello,")

fun greet(name: String) {
    text.replace("{name}", name)
        .let { correctStyle(it) }
        .capitalize()
        .let { print(it) }
}

// Usage
greet("reader") // Prints: Hello, reader
```

除此之外，还可作为参数传递函数索引，以实现更简单的语法，如下所示：

```
text.replace("{name}", name)
   .let(::correctStyle)
   .capitalize()
   .let(::print)
```

2. 针对初始化操作使用 apply 函数

某些时候，可通过调用相关方法或调整一些属性，进而创建或初始化对象。例如，下

列代码将生成 Button：

```
val button = Button(context)
button.text = "Click me"
button.isVisible = true
button.setOnClickListener { /* ... */ }
this.button = button
```

利用 apply 扩展函数，可减少相应的冗余代码。在 button 定义为接收者对象的地方，可根据当前上下文调用所有此类方法，如下所示：

```
button = Button(context).apply {
    text = "Click me"
    isVisible = true
    setOnClickListener { /* ... */ }
}
```

3. also 函数

also 函数类似于 apply，其差别在于，参数函数作为参数被接收，而非接收者。当需要在某个对象上执行操作（但并非是初始化行为）时，则推荐使用该方法，如下所示：

```
abstract class Provider<T> {

   var original: T? = null
   var override: T? = null

   abstract fun create(): T

   fun get(): T = override ?: original ?: create().also { original = it }
}
```

当需要在处理过程中执行某些操作时，推荐使用 also 函数。例如，在使用 Builder 模式的对象构造过程中，对应代码如下所示：

```
fun makeHttpClient(vararg interceptors: Interceptor) =
    OkHttpClient.Builder()
         .connectTimeout(60, TimeUnit.SECONDS)
         .readTimeout(60, TimeUnit.SECONDS)
         .also { it.interceptors().addAll(interceptors) }
         .build()
```

另一种情况则是，当已处于扩展函数中，且不需要添加另一个扩展接收者时，也推荐使用 also 函数，如下所示：

```
class Snail {
    var name: String = ""
    var type: String = ""
    fun greet() {
        println("Hello, I am $name")
    }
}

class Forest {
     var members = listOf<Sneil>()
    fun Sneil.reproduce(): Sneil = Sneil().also {
        it.name = name
        it.type = type
        members += it
    }
}
```

4. run 和 with 函数

run 和 with 函数接收包含接收者的 Lambda 字面值作为参数，并于随后返回其结果。二者间的差别在于，run 获取接收者，而 with 函数并非是扩展函数，且仅接收正在操作的对象作为参数。另外，当设置某个对象时，两个函数均可用作 apply 的替代方案，如下所示：

```
val button = findViewById(R.id.button) as Button

button.apply {
    text = "Click me"
    isVisible = true
    setOnClickListener { /* ... */ }
}

button.run {
    text = "Click me"
    isVisible = true
    setOnClickListener { /* ... */ }
}

with(button) {
    text = "Click me"
    isVisible = true
    setOnClickListener { /* ... */ }
}
```

apply、run 和 with 间的差异主要体现在：apply 返回接收者对象，而 run 和 with 则返回函数字面值结果。虽然每个函数均十分有用，但在某些情况下读者需对此有所选择。若无须任何返回值时，究竟使用哪个函数目前尚存在争议。通常情况下，建议使用 run 或 with 函数，其原因在于，also 函数常用于需要使用到返回值这一类场景。

run 和 with 间的差异可解释为：当某个值可空时，推荐使用 run 函数，而非 with 函数——可采用安全调用或非空断言，如下所示：

```
val button = findViewById(R.id.button) as? Button

button?.run {
    text = "Click me"
    isVisible = true
   setOnClickListener { /* ... */ }
}
```

相应地，若表达式较短，则推荐使用 with 函数，如下所示：

```
val button = findViewById(R.id.button) as Button

with(button) {
    text = "Click me"
    isVisible = true
    setOnClickListener { /* ... */ }
}
```

另外，若表达式较长，则推荐使用 run（而非 with），如下所示：

```
itemAdapter.holder.button.run {
    text = "Click me"
    isVisible = true
    setOnClickListener { /* ... */ }
}
```

5. to 函数

第 4 章曾讨论了中缀函数，不仅可定义为成员函数，还可确定为扩展函数。相应地，可针对任意对象生成中缀扩展函数。to 函数即是此类扩展函数中的一种，第 2 章曾对此进行了简要介绍，下面将考察其具体实现，对应的定义如下所示：

```
infix fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)
```

据此，可将 to 置于两个对象之间，并通过下列方式与 Pair 结合使用：

```
println( 1 to 2 == Pair(1, 2) ) // Prints: true
```

注意，通过定义 infix 扩展函数，可将该函数指定为任意类型的参数，如下所示：

```
infix fun <T> List<T>.intersection(other: List<T>)
    = filter { it in other }

listOf(1, 2, 3) intersection listOf(2, 3, 4) // [2,3]
```

7.5.2　特定领域内的语言

某些特性可定义类型安全的构造器，例如 Lambda 字面值以及成员扩展函数（源自 Groovy）。其中，较为有名的 Android 示例是 Gradle 配置，即 build.gradle，目前采用 Groovy 编写。这一类构造器可视作 XML、HTML 或配置文件的较好的替代方案。Kotlin 的优点在于：可确保此类配置类型安全，并提供了较好的 IDE。此类构造器是 Kotlin 特定领域语言（DSL）的一个示例。

在 Android 中，较为常见的 Kotlin DSL 是可选回调类的实现，并用于解决以下问题：缺少回调接口（包含多个方法）的函数式支持。对此，经典的实现方式将使用到对象表达，如下所示：

```
searchView.addTextChangedListener(object : TextWatcher {
  override fun beforeTextChanged(s: CharSequence, start: Int, count: Int,
after: Int) {}

  override fun onTextChanged(s: CharSequence, start: Int, before: Int,
count: Int) {
    presenter.onSearchChanged(s.toString())
  }

  override fun afterTextChanged(s: Editable) {}
})
```

上述实现的主要问题包括以下几点内容：

- 须实现接口中的全部方法。
- 须针对各个方法实现函数结构。
- 须使用对象表达。

下列代码定义了该类，并将回调作为可变属性：

```
class TextWatcherConfig : TextWatcher {

  private var beforeTextChangedCallback: (BeforeTextChangedFunction)? =
```

```
null // 1
  private var onTextChangedCallback: (OnTextChangedFunction)? = null // 1
  private var afterTextChangedCallback: (AfterTextChangedFunction)? = null
// 1

  fun beforeTextChanged(callback: BeforeTextChangedFunction){ // 2
    beforeTextChangedCallback = callback
}

  fun onTextChanged(callback: OnTextChangedFunction) { // 2
    onTextChangedCallback = callback
  }

  fun afterTextChanged(callback: AfterTextChangedFunction) { // 2
    afterTextChangedCallback = callback
}

  override fun beforeTextChanged (s: CharSequence?, start: Int, count: Int,
  after: Int) { // 3
    beforeTextChangedCallback?.invoke(s?.toString(), start, count, after)
// 4
}

  override fun onTextChanged(s: CharSequence?, start: Int, before:
  Int, count: Int) { // 3
    onTextChangedCallback?.invoke(s?.toString(), start, before, count) // 4
  }

  override fun afterTextChanged(s: Editable?) { // 3
    afterTextChangedCallback?.invoke(s)
  }
}

private typealias BeforeTextChangedFunction =
(text: String?, start: Int, count: Int, after: Int)->Unit
private typealias OnTextChangedFunction =
(text: String?, start: Int, before: Int, count: Int)->Unit

private typealias AfterTextChangedFunction =
(s: Editable?)->Unit
```

针对注释 1，当重载函数被调用时，将使用回调；对于注释 2，函数用于设置新的回调，其名称对应于处理程序函数名，但将回调作为参数；对于注释 3，各事件处理程序函数调用回调（若存在）；对于注释 4，为了进一步简化应用，还须对类型进行调整。原始方法中的 CharSequence 被修改为 String。

当前，全部任务集中于扩展函数，并用于简化回调配置。注意，其名称不可等同于 TextView 的名称。全部工作仅须稍作调整即可，如下所示：

```
fun TextView.addOnTextChangedListener(config: TextWatcherConfig.()->Unit)
{
   val textWatcher = TextWatcherConfig()
   textWatcher.config()
   addTextChangedListener(textWatcher)
}
```

根据这一定义，可通过下列方式定义所需回调，如下所示：

```
searchView.addOnTextChangedListener {
   onTextChanged { text, start, before, count ->
       presenter.onSearchChanged(text)
   }
}
```

这里，可采用下划线隐藏未用参数，并对实现过程加以改进，如下所示：

```
searchView.addOnTextChangedListener {
    onTextChanged { text, _, _, _ ->
        presenter.onSearchChanged(text)
    }
}
```

当前忽略了两个其他回调，即 beforeTextChanged 和 afterTextChanged，但仍可添加其他实现，如下所示：

```
searchView.addOnTextChangedListener {
    beforeTextChanged { _, _, _, _ ->
        Log.i(TAG, "beforeTextChanged invoked")
    }
    onTextChanged { text, _, _, _ ->
        presenter.onSearchChanged(text)
    }
    afterTextChanged {
        Log.i(TAG, "beforeTextChanged invoked")
    }
}
```

通过该方式定义的监听器包含下列属性：

- 相比于对象表达实现更加简短。
- 涵盖了默认函数实现。
- 可隐藏未用参数。

虽然 Android SDK 中存在多种包含多个处理程序的监听器，但可选回调类的 DSL 实现在 Android 中则更为常用。另外，库中还包含了类似的实现，例如之前提及的 Anko。

DSL 用于定义布局结构，且无须使用 XML 布局文件。下面将定义一个函数，添加并配置 LinearLayout 和 TextView，并以此确定简单的视图，如下所示：

```
  fun Context.linearLayout(init: LinearLayout.() -> Unit): LinearLay out {
      val layout = LinearLayout(this)
      layout.layoutParams = LayoutParams(WRAP CONTENT, WRAP CONTENT)
      layout.init()
      return layout
  }

  fun ViewGroup.linearLayout(init: LinearLayout.() -> Unit): LinearLayout
{
      val layout = LinearLayout(context)
      layout.layoutParams = LayoutParams(WRAP CONTENT, WRAP CONTENT)
      layout.init()
      addView(layout)
      return layout
  }

  fun ViewGroup.textView(init: TextView.() -> Unit): TextView {
      val layout = TextView(context)
      layout.layoutParams = LayoutParams(WRAP CONTENT, WRAP CONTENT)
      layout.init()
      addView(layout)
      return layout
  }

  // Usage
  class MainActivity : AppCompatActivity() {

     override fun onCreate(savedInstanceState: Bundle?) {
         super.onCreate(savedInstanceState)
         val view = linearLayout {
             orientation = LinearLayout.VERTICAL
             linearLayout {
```

```
                orientation = LinearLayout.HORIZONTAL
                textView { text = "A" }
                textView { text = "B" }
            }
            linearLayout {
                orientation = LinearLayout.HORIZONTAL
                textView { text = "C" }
                textView { text = "D" }
            }
        }
        setContentView(view)
    }
}
```

另外，还可从头开始自定义 DSL，下面将定义一个简单的 DSL 进而确定文章列表。如前所述，每一篇文章应在不同的分类中定义，并包含自身的名称、URL 和标签。相关定义如下所示：

```
category("Kotlin") {
    post {
        name = "Awesome delegates"
        url = "SomeUrl.com"
    }
    post {
        name = "Awesome extensions"
        url = "SomeUrl.com"
    }
}
category("Android") {
    post {
         name = "Awesome app"
         url = "SomeUrl.com"
         tags = listOf("Kotlin", "Google Login")
    }
}
```

此处，最为简单的对象是 Post 类，该类加载了跟帖属性并可在后续操作中予以修改，如下所示：

```
class Post {
    var name: String = ""
    var url: String = ""
    var tags: List<String> = listOf()
}
```

随后，还须定义加载当前分类的数据类，进而存储跟帖列表及其名称。除此之外，还须定义一个函数用以添加简单的帖子。该函数须包含一个函数参数，其中，Post 表示为接收者类型。对应定义如下所示：

```
class PostCategory(val name: String) {
    var posts: List<Post> = listOf()

    fun post(init: Post.()->Unit) {
        val post = Post()
        post.init()
        posts += post
    }
}
```

另外，还应定义一个类加载分类列表，并支持简单的分类定义，如下所示：

```
class PostList {

    var categories: List<PostCategory> = listOf()

    fun category(name: String, init: PostCategory.()->Unit) {
        val category = PostCategory(name)
        category.init()
        categories += category
    }
}
```

当前，全部所需任务集中于 **definePosts** 函数，其定义如下所示：

```
fun definePosts(init: PostList.()->Unit): PostList {
    val postList = PostList()
    postList.init()
    return postList
}
```

至此，可通过简单、类型安全的构造器定义对象结构，如下所示：

```
  val postList = definePosts {
     category("Kotlin") {
         post {
             name = "Awesome delegates"
             url = "SomeUrl.com"
         }
         post {
             name = "Awesome extensions"
```

```
                url = "SomeUrl.com"
            }
        }
        category("Android") {
            post {
                name = "Awesome app"
                url = "SomeUrl.com"
                tags = listOf("Kotlin", "Google Login")
            }
        }
}
```

DSL 是一类功能强大的概念，越来越多地用于 Kotlin 社区中。同时，Kotlin 已完全替代了下列内容：

- Android 布局文件（Anko）。
- Gradle 配置文件。
- HTML 文件（kotlinx.html）。
- JSON 文件（Kotson）。

下面讨论相关的示例库，定义 Kotlin DSL 进而提供类型安全的构造器。

Anko 提供了 DSL 进而定义 Android 视图，且无须使用 XML 布局。这与之前讨论的某些示例十分相似，但 Anko 可完全替代项目中的 XML 布局文件。下列代码显示了 Anko DSL 中编写的视图示例：

```
verticalLayout {
    val name = editText()
    button("Say Hello") {
        onClick { toast("Hello, ${name.text}!") }
    }
}
```

对应结果如图 7.8 所示（图像来源于 https://github.com/Kotlin/anko）。

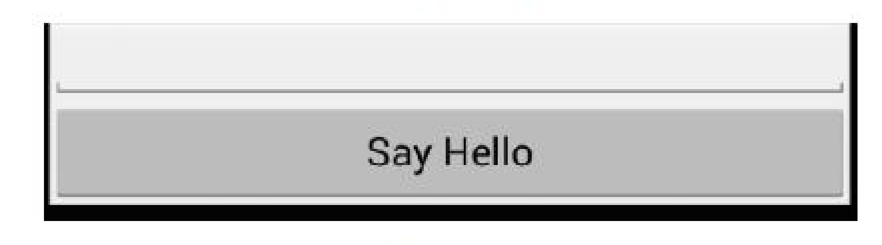


图 7.8

通过 Anko DSL，还可定义更加复杂的布局，此类视图可置于自定义类中以作为视图使用，或者直接置于 onCreate 方法中，如下所示：

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

   verticalLayout {
       padding = dip(30)
       editText {
           hint = "Name"
           textSize = 24f
       }
       editText {
            hint = "Password"
            textSize = 24f
       }
       button("Login") {
          textSize = 26f
       }
   }
}
```



读者可访问 Anko Wiki 以获取更多内容，对应网址为 https://github.com/Kotlin/anko/wiki/ Anko-Layouts。

关于 DSL 布局定义是否可替代 XML 定义，目前尚存争议。在本书编写时，此类视图定义方式较少使用。目前尚缺少 Google 的支持。但 Google 宣布将对 Kotlin 予以支持。因此，这一概念将会变得更加流行；DSL 布局也将赢得更多的支持，甚至有可能成为通用定义。

7.6 本章小结

本章讨论了 Kotlin 中的扩展函数和属性，二者均定义为顶级和类型成员。另外，本章还介绍了 Kotlin 标准库扩展函数的使用方式，进而可简化集合处理并执行各项操作。同时也考察了基于接收者的函数类型，以及基于接收者的函数字面值。一些较为重要的标准库泛型函数（采用了扩展）包括 let、apply、also、with、run 以及 to。最后，本章还探讨了 Kotlin 中 DSL 的定义方式以及应用场合。

第 8 章将讨论 Java 中未涉及的特性，此类特性在 Kotlin 开发中则十分常见，即类和属性的委托。

第 8 章　委 托 机 制

Kotlin 从设计模式中借鉴了大量内容。前述章节曾讨论了如何使用单例模式简化对象声明，以及观察者模式的简化方式（依据高阶函数和函数类型）。除此之外，借助于 Lambda 表达式和函数类型，Kotlin 还简化了大多数函数模式应用。本章将考察基于类委托的委托模式和装饰器模式。除此之外，本章还将介绍一种全新的特性——属性委托，及其使用方式，进而增强 Kotlin 中的属性功能。

本章主要涉及以下内容：

- 委托模式。
- 类委托。
- 装饰器模式。
- 属性委托。
- 源自标准库的属性委托。
- 构建自定义属性委托。

8.1　类　委　托

Kotlin 中包含了一种称之为类委托的特性，该特性并不显著，但其实际应用范围则较广。注意，类委托与两种设计模式关系紧密，即委托模式和装饰器模式。下面将详细讨论这两种模式。委托模式和装饰器模式已经存在了很长时间，在 Java 中，其实现需要使用到大量的样板代码。对于此类模式，Kotlin 则提供了本地支持，并将样板代码降至最低。

8.1.1　委托模式

在面向对象程序设计中，委托模式是继承的一种替代方案。也就是说，对象通过将其委托至另一个对象（委托）进而处理某个请求，而不是扩展该类。

为了支持 Java 中的多态机制，两个对象须实现在全部委托方法和属性的同一接口。下列代码展示了委托模式的简单示例。

```
interface Player { // 1
    fun playGame()
}

class RpgGamePlayer(val enemy: String) : Player {
```

```
    override fun playGame() {
        println("Killing $enemy")
    }
}

class WitcherPlayer(enemy: String) : Player {
    val player = RpgGamePlayer(enemy) // 2
    override fun playGame() {
        player.playGame() // 3
    }
}

// Usage
RpgGamePlayer("monsters").playGame() // Prints: Killing monsters
WitcherPlayer("monsters").playGame() // Prints: Killing monsters
```

针对注释 1，当考察类委托时，须设置一个接口并定义委托方法；在注释 2 中，设置了委托的对象；对于注释 3，WitcherPlayer 中的全部方法应调用委托对象（player）上的对应方法。

由于 WitcherPlayer 类将定义于 Player 接口中的方法委托至 RpgGamePlayer (player)类型接口。通过继承机制（而非委托）也可实现类似的结果，如下所示：

```
class WitcherPlayer() : RpgGamePlayer()
```

初看之下，两种方案彼此相似，但实际上，委托和继承机制包含了许多差异。一方面，继承机制在 Java 中更为常见，且与多种 OOP 模式关系紧密。另外一方面，委托机制也涉及了大量内容，例如 Gang of Four 编写的 *Design Patterns* 一书便极具影响力。书中指出：优先使用对象组合，而非继承；而在另一本知名著作 *Effective Java* 中则提到：优先使用组合，而非继承（第 6 项）。由此可见，二者均对委托机制提出了强有力的支持。委托模式下的基本观点包括：

- 通常情况下，类针对继承机制而设计。当覆写方法时，人们通常不会关注与类内部行为相关的底层假设条件（当方法被调用时，该调用对对象、状态等的影响方式）。例如，当覆写某个方法时，可能并不会意识到该方法还将被其他方法所使用。因此，覆写后的方法可能被超类以一种不确定的方式被调用。即使对该方法调用进行检测，该行为仍会导致在当前类的某个新版本中产生变化（例如，从外部库中扩展类），从而破坏子类的行为。针对继承，一般会适当涉及少量的类，但几乎所有非抽象类针对具体应用（包括委托）而加以设计。
- 在 Java 中，可能会将某个类托管至多个类中，但仅从一个类中继承。
- 通过接口，可指定需要托管的方法和属性，这与接口分离机制保持一致——不可向

客户端暴露不必要的方法。

- 某些类会定义为 final，因而仅可委托至其中。实际上，未针对继承设计的所有类均应为 final。Kotlin 设计者已认识到这一问题，默认条件下，Kotlin 中的全部类均为 final。
- 针对公有库，将某个类指定为 final 并提供相应的接口可视为一种较好的操作方法。对此，可调整类实现，且无须担心会影响到库用户（从接口角度来看，只要行为保持一致即可）。因此，这也使得继承操作不再可行，但仍是委托目标的最佳候选。

关于类的设计方式进而支持继承机制，以及何时应使用委托，读者可参考 *Effective Java* 一书中的第 16 项：优先使用组合，而非继承。

当然，委托机制也存在某些缺点，其中包括：

- 需要创建接口，以指定应委托的方法。
- 无法访问受保护的方法和属性。

在 Java 中，关于使用继承问题，仍存在某些激烈的争论——继承更易于实现。对此，可比较 WitcherPlayer 示例代码，不难发现，委托中涵盖了较多的附加代码，如下所示：

```
class WitcherPlayer(enemy: String) : Player {
    val player = RpgGamePlayer(enemy)
    override fun playGame() {
        player.playGame()
    }
}

class WitcherPlayer() : RpgGamePlayer()
```

当处理包含多个接口的方法时，这将产生问题。然而，现代编程语言均十分重视委托设计模式，大多数语言设置了本地类委托支持。例如，Swift 和 Groovy 对委托模式均提供了强大的支持；Ruby 语言则通过其他机制对此予以支持，并简化了该模式的应用，同时使得样板代码数量降至最低。这里，示例中的 WitcherPlayer 类在 Kotlin 中可通过下列方式实现：

```
class WitcherPlayer(enemy: String) : Player by RpgGamePlayer(enemy) {}
```

当使用 by 关键字时，即通知编译器，将定义于 Player 接口中的全部方法从 Witcher Player 委托至 RpgGamePlayer。RpgGamePlayer 将在 WitcherPlayer 构造过程中创建。简而言之，WitcherPlayer 将定义于 Player 接口中的方法委托至新的 RpgGamePlayer 对象中。

在编译器过程中，Kotlin 编译器将生成 WitcherPlayer 中源自 Player 的、未实现的方法，并利用 RpgGamePlayer 实例调用对其填充（该方式与第一个示例中的实现方式相同）。这

里体现了一处重大改进：无须亲自实现此类方法。另外须注意的是，如果委托方法签名发生变化，无须修改委托至其中的全部对象，因而对应类易于维护。

除此之外，还存在另一种委托实例的创建和加载方式，并通过构造函数予以提供，如下所示：

```
class WitcherPlayer(player: Player) : Player by player
```

同时，还可委托至定义于构造函数中的属性中，如下所示：

```
class WitcherPlayer(val player: Player) : Player by player
```

最后，还可在类声明时委托至任意可访问的属性中，如下所示：

```
val d = RpgGamePlayer(10)
class WitcherPlayer(a: Player) : Player by d
```

而且，一个对象可包含多个不同的委托，如下所示：

```
interface Player {
    fun playGame()
}

interface GameMaker { // 1
    fun developGame()
}

class WitcherPlayer(val enemy: String) : Player {
    override fun playGame() {
       print("Killin $enemy! ")
    }
}

class WitcherCreator(val gameName: String) : GameMaker{
    override fun developGame() {
       println("Makin $gameName! ")
    }
}

class WitcherPassionate :
    Player by WitcherPlayer("monsters"),
    GameMaker by WitcherCreator("Witcher 3") {

    fun fulfillYourDestiny() {
        playGame()
        developGame()
```

```
    }
}

// Usage
WitcherPassionate().fulfillYourDestiny() // Killin monsters! Makin
Witcher 3!
```

针对注释 1，WitcherPlayer 类将 Player 接口委托至新的 RpgGamePlayer 对象中，将 GameMaker 委托至 WitcherCreator 对象中，另外还包含了 fulfillYourDestiny 函数（该函数使用了源自两个委托中的函数）。注意，WitcherPlayer 和 WitcherCreator 均未标记为 open，虽然可被委托，但无法被扩展。

通过编程语言方面的支持，与继承机制相比，委托更具吸引力。尽管此类模式包含相应的优缺点，但读者应了解何时使用这一类模式。下列内容列举了一些委托的应用场合：

- 当子类与里氏替换规则冲突时，例如，继承仅实现为复用超类代码时，但实际行为并不与其相符。
- 当子类仅使用超类中的部分方法时。其中，可能会调用一个不应该调用的超类方法。当采用委托机制时，可复用所选取的方法（定义于当前接口中）。
- 当无法或不应使用继承时，其中包括：
  ★ 类定义为 final。
  ★ 相对于接口无法访问或使用。
  ★ 未针对继承而设计。

注意，默认状态下 Kotlin 中的类表示为 final，如果将其置于某个库中，则很可能无法修改或打开该类；而委托仅可视作是一个选项，进而创建包含不同行为的类。

里氏替换规则是 OOP 程序设计中的概念，其中，全部子类的行为都应该像它们的超类一样。简单地讲，如果某个类通过单元测试，则其子类也应通过该测试。该原则由 Robert C. Martin 提出，并作为较为重要的编程规则之一，具体内容可参考 *Cean Code* 一书。

Effective Java 一书中曾讲到：仅当子类真正地表示为超类的子类型时，继承机制方为适宜。换而言之，仅当两个类之间存在 is 关系时，类 B 方可扩展某个类。如果视图令类 B 扩展类 A，则应首先询问自己：每个 B 是否真正是 A？稍后将介绍，在其他情况下，建议使用组合机制。

值得注意的是，Cocoa（Apple 的 UI 框架，用以构建软件程序并在 iOS 上运行）常采用委托，而非继承机制。该模式变得越来越流行，Kotlin 也对此提供了强大的支持。

8.1.2 装饰器模式

在 Kotlin 中，类委托机制的另一个用武之地是在实现装饰器模式时。装饰器模式（也称作包装器模式）可向现有类中加入某种行为，且无须采用继承操作。相比于扩展（可添加新行为，且不需要调整对象），此处将构建一个包含不同行为的具体对象。装饰器模式使用了委托机制，但却采取了一种较为特殊的方式——委托来自于类的外部。图 8.1 显示了经典的 UML 图。

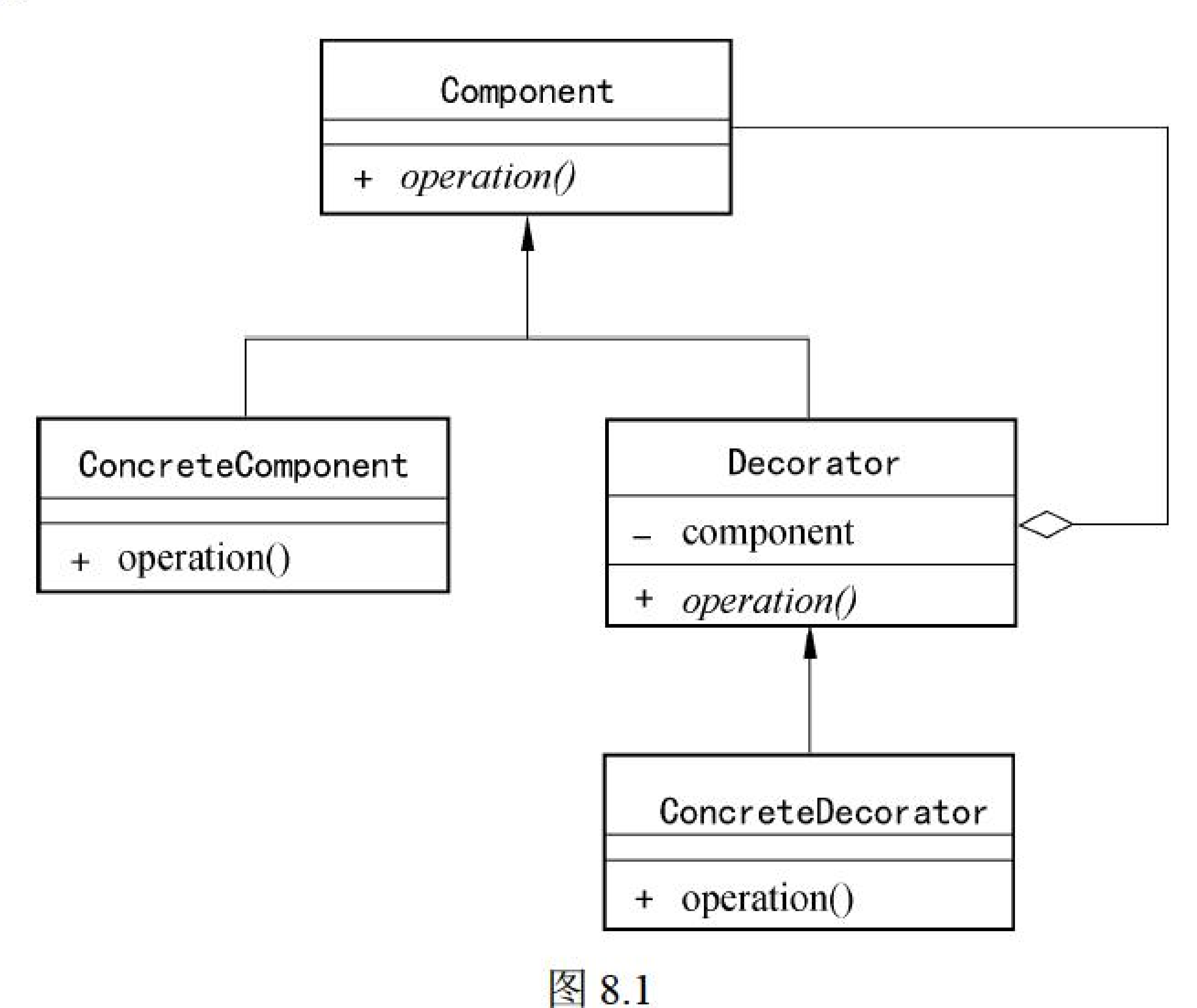


图 8.1

装饰器中包含了所装饰的对象，同时实现了同一接口。

InputStream 则是 Java 中较为常见的装饰器示例。对此，存在不同类型扩展了 InputStream，同时存在多种装饰器可用于向其添加功能项。该装饰器可用于添加缓冲区、获取压缩文件中的内容，或者将文件内容转换为 Java 对象。在下列示例中，使用了多个装饰器以读取压缩 Java 对象。

```
// Java
FileInputStream fis = new FileInputStream("/someFile.gz"); // 1
BufferedInputStream bis = new BufferedInputStream(fis); // 2
GzipInputStream gis = new GzipInputStream(bis); // 3
ObjectInputStream ois = new ObjectInputStream(gis); // 4
SomeObject someObject = (SomeObject) ois.readObject(); // 5
```

对于注释 1，针对文件读取创建一个简单的流；针对注释 2，生成一个包含缓冲区的新流；针对注释 3，生成一个新的流，其中包含了读取 GZIP 文件格式的压缩数据；针对注释 4，

生成一个新流并添加一个功能项，用于反序列化之前采用 ObjectOutputStream 编写的基本数据和对象；针对注释 5，流用于 ObjectInputStream 的 readObject 方法中，但该示例中的全部对象均实现于 InputStream 中（因而有可能通过这一方式对其进行打包），并可通过当前接口所定义的方法所读取。

需要注意的是，上述模式与继承类似，但却可确定所用的装饰器及其顺序，因而在应用过程中体现了更大的灵活性以及更多的可能性。一些人认为，如果设计者定义一个包含全部设计功能的大型类，随后使用相关方法开启或关闭其中的某些功能，将会更好地运用 InputStream。实际上，该方案违背了单一职责原则，从而导致代码更加复杂且难以扩展。

在实际操作过程中，虽然装饰器模式可视作一种最佳方案，但在 Java 项目中却较少使用，其原因在于：具体实现过程相对复杂。接口中一般涵盖多个方法，为其在每个装饰器中创建一个委托，将产生大量的样板代码。在 Kotlin 中，情况则有所不同——之前曾有所讨论，Kotlin 中的类委托机制实际上十分简单。下面考察装饰器模式中类委托机制的某些经典示例，假设需要向多个不同的 ListAdapter 添加第一个位置作为 0 元素。这一额外的位置包含了某些特定属性。对此，无法采用继承机制予以实现——针对不同列表的 ListAdapter 具有不同的类型（这也是一类标准情形）。此处，可调整各个类的行为（DRY 规则），或者生成一个装饰器。该装饰器的简短代码如下所示：

```
class ZeroElementListDecorator(val arrayAdapter: ListAdapter) :
    ListAdapter by arrayAdapter {
  override fun getCount(): Int = arrayAdapter.count + 1
  override fun getItem(position: Int): Any? = when {
      position == 0 -> null
      else -> arrayAdapter.getItem(position - 1)
  }

  override fun getView(position: Int, convertView: View?,parent:
ViewGroup): View = when {
    position == 0 -> parent.context.inflator
        .inflate(R.layout.null element layout, parent, false)
    else -> arrayAdapter.getView(position - 1, convertView, parent)
  }
}

override fun getItemId(position: Int): Long = when {
  position == 0 -> 0
  else -> arrayAdapter.getItemId(position - 1)
}
```

此处使用了 Context 中的 inflator 扩展属性，该属性通常包含于 Kotlin Android 项目中，相关内容可参考第 7 章。对应代码如下所示：

```
val Context.inflater: LayoutInflater
    get() = LayoutInflater.from(this)
```

通过该方式定义的 ZeroElementListDecorator 类将添加包含静态视图的第一个元素，其应用方式如下所示：

```
val arrayList = findViewById(R.id.list) as ListView
val list = listOf("A", "B", "C")
val arrayAdapter = ArrayAdapter(this,
      android.R.layout.simple list item 1, list)
arrayList.adapter = ZeroElementListDecorator(arrayAdapter)
```

在 ZeroElementListDecorator 中，代码看上去稍显复杂，因而需要覆写 4 个方法。但实际上，鉴于 Kotlin 的委托机制，至少存在 8 个方法且无须覆写。不难发现，Kotlin 的类委托机制大大简化了装饰器模式的实现过程。

装饰器模式易于实现，且具有较好的直观性，同时还可用于多种不同的场合，以扩展包含附加功能的某个类。鉴于其安全性，因而该模式具备较好的安全特性，并作为一类良好的操作加以引用。此类示例仅体现了类委托的部分可能性，更多的、基于所述模式的用例，以及类委托机制的应用，尚等待读者进一步发现，以使项目更加清晰、安全、简洁。

8.2　属性委托

除了类委托之外，Kotlin 还支持属性委托。本节主要讨论属性委托的含义，考察 Kotlin 标准库中的属性委托机制，并学习如何创建和使用自定义属性委托。

8.2.1　属性委托的含义

下面首先解释属性委托的含义，其应用示例如下所示：

```
class User(val name: String, val surname: String)

var user: User by UserDelegate() // 1

println(user.name)
user = User("Marcin","Moskala")
```

对于注释 1，此处将 user 属性委托至 UserDelegate 的实例（用构造函数创建）中。

属性委托与类委托有几分相似，通过相同的关键字（by），可委托至某个对象中。某属性的每次调用（set/get）将委托至另一个对象（UserDelegate）。通过这一方式，可针对多个属性复用同一行为。例如，仅当满足某种场合时设置属性值；或者在属性被访问/更新时添加日志项。

实际上，属性并不需要幕后字段（backing field），可能仅通过 getter（只读）或 getter/setter（读-写）加以定义。从底层角度来看，属性委托仅被转换为对应的方法调用（setValue/getValue）。此时，上述示例将被编译为下列代码：

```
var p$delegate = UserDelegate()
var user: User
get() = p$delegate.getValue(this, ::user)
set(value) {
    p$delegate.setValue(this, ::user, value)
}
```

该示例表明，通过使用 by 关键字，将 setter 和 getter 调用委托至 delegate。这也解释了包含 getValue 和 setValue 函数（设置了正确的参数，稍后将对此加以讨论）的任意对象可用作委托（针对只读属性 getValue 已然足够，由于此处仅需要 getter）。重要的是，须用作属性委托的全部类将包含这两个方法，同时无须使用到接口。UserDelegate 的实现示例如下所示：

```
class UserDelegate {
    operator fun getValue(thisRef: Any?, property: KProperty<*>):
          User = readUserFromFile()
    operator fun setValue(thisRef: Any?, property: KProperty<*>,
          user:User) {
        saveUserToFile(user)
    }
    //...
}
```

setValue 和 getValue 由于设置和获取属性值（属性的 setter 调用委托至 setValue 方法；属性 getter 将数值委托至 getValue 方法）。两个函数均需要采用 operator 关键字，同时包含了一些特定的参数集合，这些参数决定委托的位置和属性。如果某个属性表示为只读，那么对象仅须包含一个 getValue 方法并用作其委托，如下所示：

```
class UserDelegate {

    operator fun getValue(thisRef: Any?, property: KProperty<*>):
```

```
    User = readUserFromFile()
}
```

getValue 方法的返回类型以及属性的类型（用户定义于 setValue 方法中）决定了委托属性的类型。

getValue 和 setValue 函数的第一个参数（thisRef）包含了委托所用的上下文的引用，并可用于委托所用的类型。例如，可通过下列方式定义仅在 Activity 类中使用的委托：

```
class UserDelegate {
    operator fun getValue(thisRef: Activity, property: KProperty<*>):
          User = thisRef.intent
          .getParcelableExtra("com.example.UserKey")
}
```

从中可以看到，此处存在一个全部上下文提供的 this 引用。仅在扩展函数内部，或扩展属性上才存在 null 值。指向 this 的引用用于获取某些来自上下文的数据。如果将其设置为 Activity 类型，则可仅在 Activity 中使用该委托（在 this 为 Activity 的任意上下文中）。

除此之外，若需要强制当前委托仅用于顶级位置，则可将第一个参数（thisRef）的类型指定为 Nothing?——该类型的唯一可能值为 null。

上述方法中的另一个参数为 property，包含了指向委托属性的一个引用，其中涵盖了其元数据（属性名以及类型等）。

属性委托可用于定义于任意上下文中的属性（顶级属性、成员属性以及局部属性等），如下所示：

```
var a by SomeDelegate() // 1

fun someTopLevelFun() {
    var b by SomeDelegate() // 2
}

class SomeClass() {
    var c by SomeDelegate() // 3

    fun someMethod() {
        val d by SomeDelegate() // 4
    }
}
```

对于注释 1，基于委托的顶级属性；对于注释 2，包含委托的局部属性（在顶级函数中）；对于注释 3，包含委托的成员属性。

后续章节将讨论 Kotlin 标准库中的委托机制，并体现了其有效性以及属性委托的应用

方式。

8.2.2　预定义委托

Kotlin 标准库包含了一些属性委托，且使用起来十分方便。下面讨论其在实际项目中的使用方式。

1. lazy 函数

某些时候，需要初始化某个对象，且需要确保该对象仅初始化一次，以供首次使用。在 Java 中，可通过下列方式处理这一问题：

```
private var  someProperty: SomeType? = null
private val somePropertyLock = Any()
val someProperty: SomeType
get() {
    synchronized(somePropertyLock) {
        if ( someProperty == null) {
            someProperty = SomeType()
        }
        return  someProperty!!
    }
}
```

这一构造过程是 Java 开发中的常见模式。通过延迟（lazy）委托，Kotlin 可采用较为简单的方式处理此类问题，这也是较为常用的委托，且仅与只读属性（val）协同工作，对应的应用方式如下所示：

```
val someProperty by lazy { SomeType() }
```

标准库中的 lazy 函数用于提供委托，如下所示：

```
public fun <T> lazy(initializer: () -> T):
      Lazy<T> =  SynchronizedLazyImpl(initializer)
```

在 SynchronizedLazyImpl 的示例对象中，该对象用作属性委托。大多数时候，它被称作源自其对应函数名的延迟委托。其他委托名称与提供它们的函数名称相同。



延迟委托同样包含了线程安全机制。默认状态下，委托具备完整的线程安全特性；但若已经知晓仅存在单一线程，则可进行适当调整，以使函数更加高效，为了关闭线程安全机制，需要将 enum 类型值 LazyThreadSafetyMode.NONE 作为 lazy 函数的第一个参数，如下所示：

```
val someProperty by lazy(LazyThreadSafetyMode.NONE) {
SomeType() }
```

鉴于延迟委托，属性的初始化操作将被延迟，直至需要使用到对应值时。延迟委托应用涵盖了多个优点，其中包括：

- 快速的类初始化可缩短应用程序的启动时间——数值初始化将被延迟，直至其第一次使用时。
- 某些操作流不会使用到相关值，因而无须对其进行初始化——这将极大地节省资源（包括内存、处理器时间以及电量）。

除此之外，某些对象还可在类实例创建后生成。例如，在 Activity 中，在使用 set ContentView 方法设置布局之前，将无法访问资源——一般将在 onCreate 方法中被调用。下面将考察包含视图引用元素的 Java 类，如下所示：

```
// Java
public class MainActivity extends Activity {

    TextView questionLabelView
    EditText answerLabelView
    Button confirmButtonView

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity main);

        questionLabelView = findViewById<TextView>
              (R.id.main question label);
        answerLabelView = findViewById<EditText>
              (R.id.main answer label);
        confirmButtonView = findViewById<Button>
              (R.id.main button confirm);
    }
}
```

若将其逐一转换至 Kotlin 中，对应代码如下所示：

```
class MainActivity : Activity() {

    var questionLabelView: TextView? = null
    var answerLabelView: TextView? = null
    var confirmButtonView: Button? = null
```

```
    override fun onCreate(savedInstanceState: Bundle) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.main activity)

        questionLabelView = findViewById<TextView>
              (R.id.main question label)
        answerLabelView = findViewById<TextView>
              (R.id.main answer label)
        confirmButtonView = findViewById<Button>
              (R.id.main button confirm)
    }
}
```

当采用延迟委托时，可采用更为简单的方式实现这一行为，如下所示：

```
class MainActivity : Activity() {

   val questionLabelView: TextView by lazy
{ findViewById(R.id.main question label) as TextView }
   val answerLabelView: TextView by lazy
{ findViewById(R.id.main answer label) as TextView }
   val confirmButtonView: Button by lazy
{ findViewById(R.id.main button confirm) as Button }

   override fun onCreate(savedInstanceState: Bundle) {
     super.onCreate(savedInstanceState)
     setContentView(R.layout.main activity)
   }
}
```

该方案的优点在于：

- 属性集中于一处进行声明和初始化，因而代码将更加简洁。
- 属性定义为非空类型，这可消除大量的空检测操作。
- 属性具有只读特征，因而具有线程同步或智能转换所蕴含的各种优点。
- 传递至延迟委托（包含 findViewById）的 Lambda 仅在属性首次被访问时执行。
- 与类创建过程相比，数值将于稍后被获得，这将加速启动过程。如果不打算使用此类视图，对应值将不会被获取（若视图较为复杂时，findViewById 并不是一类高效的操作方式）。
- 编译器将对未用属性进行标记。在 Java 实现中，则并不会执行该操作，因为数值集将被编译器所关注。

通过析取公共行为，并将其转换为扩展函数，可对上述实现进行改进，如下所示：

```
fun <T: View> Activity.bindView(viewId: Int) = lazy { findViewById(viewId)
as T }
```

随后，可通过更加简洁的代码定义视图绑定，如下所示：

```
class MainActivity : Activity() {

  var questionLabelView: TextView by bindView(R.id.main question label)
//1
  var answerLabelView: TextView by bindView(R.id.main answer label) // 1
  var confirmButtonView: Button by bindView(R.id.main button confirm) //
1

  override fun onCreate(savedInstanceState: Bundle) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.main activity)
  }
}
```

对于注释 1，无须设置供 bindView 函数使用的类型——可根据属性类型进行推断。

目前仅存在单一委托，当首次访问特定视图时，将于底层调用 findViewById，这可视作是一类相对简洁的方法。



针对此类问题，还存在另一种处理方式。当前较为常用的方法是 Kotlin Android 扩展插件，并在 Activities 和 Fragments 中生成视图的自动绑定。具体应用将在第 9 章加以讨论。

即使存在相关支持，绑定机制自身的优点仍十分明显。例如，可清晰地显示我们所用的视图元素；另外一点则是元素 ID 名与加载当前元素的变量名可彼此分离。除此之外，编译时间也将有所改善。

同一机制也可用于解决其他 Android 相关问题。例如，当向 Activity 传递一个参数时。对此，标准的 Java 实现方式如下所示：

```
// Java
class SettingsActivity extends Activity {

  final Doctor DOCTOR KEY = "doctorKey"
  final String TITLE KEY = "titleKey"

  Doctor doctor
  Address address
  String title

  public static void start ( Context context, Doctor doctor,
```

```
  String title ) {
    Intent intent = new Intent(context, SettingsActivity.class )
    intent.putExtra(DOCTOR_KEY, doctor)
    intent.putExtra(TITLE_KEY, title)
    context.startActivity(intent)
  }
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    doctor = getExtras().getParcelable(DOCTOR_KEY)
    title = getExtras().getString(TITLE_KEY)

    ToastHelper.toast(this, doctor.id)
    ToastHelper.toast(this, title)
  }
}
```

在 Kotlin 中，可采用相同的实现，但也可随变量声明获得参数值（getString/getParcerable）。对此，需要编写下列扩展函数：

```
fun <T : Parcelable> Activity.extra(key: String) = lazy
    { intent.extras.getParcelable<T>(key) }

fun Activity.extraString(key: String) = lazy
    { intent.extras.getString(key) }
```

随后，利用 extra 和 extraString 委托，可获取附加参数，如下所示：

```
class SettingsActivity : Activity() {

    private val doctor by extra<Doctor>(DOCTOR_KEY) // 1
    private val title by extraString(TITLE_KEY) // 1

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.settings_activity)
        toast(doctor.id) // 2
        toast(title) // 2
    }

    companion object { // 3
        const val DOCTOR_KEY = "doctorKey"
        const val TITLE_KEY = "titleKey"
```

```
    fun start(context: Context, doctor: Doctor, title: String) { // 3
        ontext.startActivity(getIntent<SettingsActivity>().apply { // 4
            putExtra(DOCTOR KEY, doctor) // 5
            putExtra(TITLE KEY, title) // 5
        })
    }
  }
}
```

对于注释 1，定义了属性，其值通过对应键从 Activity 参数中获取；对于注释 2，在 onCreate 方法中，访问源自参数的属性。当请求属性时（使用 getter），延迟委托将从附加项中获取参数，并对其存储以供后续使用；对于注释 3，定义了一个静态方法启动操作，对此需要使用伴生对象；对于注释 4，SettingsActivity::class.java 表示为 Java 类引用的对等物；对于注释 5，使用了定义于第 7 章中的相关方法。

除此之外，还可定义函数，并获取包所持有的其他类型（例如 Long 和 Serializable）。当需要实现快速编译时，对于参数注入库，例如 ActivityStarter，这可视作是一种较好的替代方案。相应地，可使用类似的函数绑定字符串、颜色、服务、资源库以及模型和逻辑的其他部分，如下所示：

```
fun <T> Activity.bindString(@IdRes id: Int): Lazy<T> =
    lazy { getString(id) }
fun <T> Activity.bindColour(@IdRes id: Int): Lazy<T> =
    lazy { getColour(id) }
```

Activity 中均为“重量型”操作，抑或取决于通过延迟委托（或异步提供的）声明的参数。另外，还应将所有元素（均依赖于须延迟初始化的元素）均定义为延迟。例如，presenter 定义依赖于 doctor 属性，如下所示：

```
val presenter by lazy { MainPresenter(this, doctor) }
```

否则，尝试构造 MainPresenter 对象将在类创建时进行，此时无法读取数据值，且无法填充 doctor 属性，因而应用程序将崩溃。

上述示例体现了 Android 项目中延迟委托的有效性，同时也提供了较好的属性委托，使代码更加简单和优雅。

2. notNull 函数

notNull 委托可视作一类最为简单的标准库委托，因而首先对其加以讨论，其应用方式

如下所示：

```
var someProperty: SomeType by notNull()
```



提供了大多数标准库委托（包括 notNull 函数）的函数均定义于 object 委托中。当对此加以使用时，需要引用该对象（Delegates.notNull()）或者予以导入(import kotlin. properties.Delegates.notNull)。假设在当前示例中，obiect 已被导入，并可省略其引用操作。

notNull 委托可将变量定义为非空类型，并在稍后予以初始化，而非在对象的构建时。此处，变量将定义为非空类型，且无须提供默认值。notNull 函数可视作 lateinit 的一种替代方案，如下所示：

```
lateinit var someProperty: SomeType
```

notNull 提供了与 lateinit 几乎相同的效果（除了错误消息之外）。当在首次设定值之前使用该属性时，将抛出一个 IllegalStateException 异常，并终止 Android 应用程序。因此，仅当已知晓某个数值在首次使用之前将被设置时方可对其加以使用。

lateinit 和 notNull 委托之间的差异也较为明显。其中，lateinit 快于 notNull 委托，因而应尽量优先使用。但此处也存在一些限制条件。例如，lateinit 无法用于基本数据，或者顶级属性。因而，在当前示例中，将采用 notNull。

下面考察 notNull 委托的实现过程，如下所示：

```
public fun <T: Any> notNull(): ReadWriteProperty<Any?, T> =
    NotNullVar()
```

不难发现，notNull 实际上表示为返回一个对象的函数，即隐藏于 ReadWriteProperty 接口背后的实际委托实例。对应的委托定义如下所示：

```
private class NotNullVar<T: Any>() : ReadWriteProperty<Any?, T> { // 1
  private var value: T? = null

  public override fun getValue(thisRef: Any?,
  property: KProperty<*>): T {
     return value ?: throw IllegalStateException("Property
            ${property.name} should be initialized before get.") // 2
  }

  public override fun setValue(thisRef: Any?,
  property: KProperty<*>, value: T) {
     this.value = value
  }
}
```

针对注释 1，类定义为 private，一种可能的原因是，该类通过 notNull 函数提供，并将其返回为 ReadWriteProperty<Any?, T>，且为 public 接口；对于注释 2，此处显示了返回值的提供方式。如果在使用期间为 null，该值不会被设置，且对应方法将抛出一个错误；否则将返回对应值。

上述委托机制易于理解。setValue 函数将数值设置为可空字段，若不为 null，则 getValue 返回该字段，否则将抛出异常。对应的错误示例如下所示：

```
var name: String by Delegates.notNull()
println(name)
// Error: Property name should be initialized before get.
```

这可视为委托属性应用的简单示例，同时也较好地介绍了属性委托的工作方式。委托属性是一种强大的构造机制，并可包含多个应用程序。

3. observable 委托

对于可变属性，observable 是一类功能强大的标准库委托。每次设置某个值时（set Value 方法将被调用），源自声明的 Lambda 函数将被调用。observable 委托的简单示例如下所示：

```
var name: String by Delegates.observable("Empty"){
    property, oldValue, newValue -> // 1
    println("$oldValue -> $newValue") // 2
}

// Usage
name = "Martin" // 3,
Prints: Empty -> Martin
name = "Igor" // 3,
Prints: Martin -> Igor
name = "Igor" // 3, 4
Prints: Igor -> Igor
```

对于注释 1，Lambda 函数参数如下所示：

- property 表示委托属性的引用，此处为名称的引用，等同于源自 setValue 和 getValue 中的属性并表示为 KProperty 类型。在当前示例（以及大多数示例）中，若未加使用，则可设置下划线符号。
- oldValue 表示 property 的上一个值（在修改前）。
- newValue 表示 property 的新值（在修改后）。

对于注释 2，在每次将新值设置至属性中时，Lambda 函数将被调用；对于注释 3，当设置

新值时，该值将被更新，但同时会调用声明于委托中的 Lambda 方法；对于注释 4，应注意每次使用 setter 时 Lambda 将被调用，新旧值之间是否相等并不重要。

需要特别引起重视的是，Lambda 在每次设置新值时将被调用，而非对象内部状态改变时，如下所示：

```
var list: MutableList<Int> by observable(mutableListOf())
{  , old, new ->
    println("List changed from $old to $new")
}

// Usage
list.add(1) // 1
list = mutableListOf(2, 3)
// 2, prints: List changed from [1] to [2, 3]
```

针对注释 1，代码并不输出任何内容——此处并未修改属性（未使用 setter），仅调整了定义于列表中的属性，而非对象自身；对于注释 2，这里修改了列表值，因而调用源自 observable 委托的 Lambda 函数，并输出文本。

observable 委托对于不可变类型来说十分有用，这与可变类型有所不同。Kotlin 中全部基本类型默认时均为不可变类型（List、Map、Set、Int、String）。下面考察 Android 中的实际操作示例，如下所示：

```
class SomeActivity : Activity() {

    var list: List<String> by Delegates.observable(emptyList()) {
        prop, old, new -> if(old != new) updateListView(new)
    }
    // ...
}
```

每次修改列表时，视图将被更新。注意，若 List 不可变，那么，在进行调整时则需要使用到 setter，从而可确保在当前操作之后列表将得到更新。与每次列表变化时调用 updateListView 方法相比，当前操作则更加简单。该模式可广泛地应用于项目中，进而声明编辑视图的属性，这将改变视图更新机制的工作方式。

另一个可通过 observable 委托处理的问题是，在 ListAdapters 中，每次列表上的元素变化时，notifyDataSetChanged 将被调用。在 Java 中，较为经典的解决方案是封装该列表，并在修改该列表的各个函数中调用 notifyDataSetChanged。在 Kotlin 中，可通过 observable 属性委托对此进行简化，如下所示：

```
var list: List<LocalDate> by observable(list) {  , old, new -> // 1
  if(new != old) notifyDataSetChanged()
}
```

对于注释 1，需要注意的是，此处的列表为不可变类型，因此，若不使用 notifyDataSet Changed 方法，则无法修改其元素。

observable 委托用于定义属性值变化上产生的行为，常用于属性每次变化时应执行的操作；或者需要利用某个视图或其他值绑定属性时。但在函数内部，则无法确定新值是否已被设置。对此，可采用 vetoable 委托。

4. vetoable 委托

vetoable 是一类标准库属性委托，其工作方式与 observable 委托类似，但包含两点主要差异，如下所示：

- 参数中的 Lambda 在新值被设置之前调用。
- 支持声明中的 Lambda 函数，以确定新值是否应被接受或丢弃。

例如，若假设列表需要包含比旧值多的数据项，则需要定义下列 vetoable 委托：

```
var list: List<String> by Delegates.vetoable(emptyList())
{  , old, new ->
   new.size > old.size
}
```

如果情况相反，即新表中不会包含比旧值多的数据项，则对应值不会发生变化。因此，可将 vetoable 视作与 observable 类似，进而可确定该值是否可被修改。假设需要一个与视图绑定的列表，但至少需要 3 个元素。此处不允许出现任何变化，进而可包含更少的元素。对应实现过程如下所示：

```
var list: List<String> by Delegates.vetoable(emptyList())
{ prop, old, new ->
    if(new.size < 3) return@vetoable false // 1
    updateListView(new)
    true // 2
}
```

对于注释 1，新表的尺寸小于 3，因而不予接受并从 Lambda 中返回 false。包含标记的 return 语句返回的 false 值（用于从 Lambda 表达式中返回）体现了一类信息，表示新值不应予以接受。

对于注释 2，Lambda 函数需要返回一个值，该值源自包含标记的 return，或者 Lambda 体的最后一行。这里，值 true 表示新值应予以接受。

对应的应用示例如下所示：

```
listVetoable = listOf("A", "B", "C") // Update A, B, C
println(listVetoable) // Prints: [A, B, C]
listVetoable = listOf("A") // Nothing happens
println(listVetoable) // Prints: [A, B, C]
listVetoable = listOf("A", "B", "C", "D", "E")
// Prints: [A, B, C, D, E]
```

除此之外，还可使其不发生任何变化，其中涉及某些其他原因，例如仍在加载数据。另外，vetoable 属性委托也可用于验证器中，如下所示：

```
var name: String by Delegates.vetoable(""){ prop, old, new ->
    if (isValid(new)) {
        showNewData(new)
        true
    } else {
        showNameError()
        false
    }
}
```

此属性仅可根据断言 isValid(new)更改为正确的值。

5. 属性委托至 Map 类型

针对包含 String 键类型的 Map 和 MutableMap，标准库包含了相应的扩展，进而提供了 getValue 和 setValue 函数。据此，map 也可用作属性委托，如下所示：

```
class User(map: Map<String, Any>) { // 1
    val name: String by map
    val kotlinProgrammer: Boolean by map
}

// Usage
val map: Map<String, Any> = mapOf( // 2
    "name" to "Marcin",
    "kotlinProgrammer" to true
)
val user = User(map) // 3
println(user.name) // Prints: Marcin
println(user.kotlinProgrammer) // Prints: true
```

对于注释 1，Map 键类型须为 String，而值类型则并无限制，通常为 Any 或 Any?；对于注

释 2，将创建包含全部值的 Map；对于注释 3，将向某个对象提供一个 map。

对于在 Map 中持有数据，这将十分有用。除此之外，还将涉及以下原因：

- 当希望简化此类数据的访问时。
- 当定义一个结构，并通知我们在当前 Map 中应接受何种类型的键。
- 当请求一个委托至 Map 中的属性时，针对等同于属性名的键，对应值将源自当前 Map。

具体实现过程如何表示？下列内容源自标准库中的一段简化代码：

```
operator fun <V, V1: V> Map<String, V>.getValue( // 1
      thisRef: Any?, // 2
      property: KProperty<*>): V1 { // 3
          val key = property.name // 4
          val value = get(key)
      if (value == null && !containsKey(key)) {
          throw NoSuchElementException("Key ${property.name}
          is missing in the map.")
      } else {
          return value as V1 // 3
      }
}
```

对于注释 1，V 表示为列表上的值类型；对于注释 2，thisRef 表示为 Any?类型。因此，Map 可用作任意上下文中的属性委托；对于注释 3，V1 表示为返回类型，并根据属性进行推断，但须为 V 类型的子类型；对于注释 4，属性名用作 map 上的键。

需要记住的是，这仅为一个扩展函数。对象表示为一个委托所需的全部工作则是涵盖一个 getValue 方法（以及针对读-写属性的 setValue 方法）。通过对象声明，甚至可从一个匿名类对象中创建一个委托，如下所示：

```
val someProperty by object { // 1
    operator fun getValue(thisRef: Any?,
    property: KProperty<*>) = "Something"
}
println(someProperty) // prints: Something
```

对于注释 1，object 未实现为接口，仅包含了一个基于适当签名的 getValue 方法。这已然可令其像只读属性委托那样工作。

注意，在 map 中，当请求属性值时，须存在一个包含此类名称的入口项，否则将抛出一个错误（令属性为可空类型仍无法改变这一状况）。

map 的委托字段十分有用。例如，当从包含动态字段的 API 中获取一个对象时。对此，

可将所提供的数据视为一个对象，进而可方便地访问其字段；另外，还须将其作为 map，并列出 API 所提供的全部字段（甚至可能会包含某些非期望字段）。

上述示例使用了 Map 且为不可变类型。因此，对象属性为只读（val）。如果需要令对象为可变类型，则应使用 MutableMap；随后，该属性可定义为可变类型（val）。对应示例如下所示：

```
class User(val map: MutableMap<String, Any>) {
    var name: String by map
    var kotlinProgrammer: Boolean by map
    override fun toString(): String = "Name: $name,
    Kotlin programmer: $kotlinProgrammer"
}

// Usage
val map = mutableMapOf( // 1
    "name" to "Marcin",
    "kotlinProgrammer" to true
)
val user = User(map)
println(user) // prints: Name: Marcin, Kotlin programmer: true
user.map.put("name", "Igor") // 1
println(user) // prints: Name: Igor, Kotlin programmer: true
user.name = "Michal" // 2
println(user) // prints: Name: Michal, Kotlin programmer: true
```

对于注释 1，属性值可通过调整 map 值进行修改；对于注释 2，属性值还可像其他属性那样被修改。实际上，对应值变化委托至 setValue，进而修改 map。

尽管这里的属性为可变类型，但仍须提供 setValue 函数。对于 MutableMap，可实现为扩展函数，其简化代码如下所示：

```
operator fun <V> MutableMap<String, V>.setValue(
    thisRef: Any?,
    property: KProperty<*>,
    value: V
) {
    put(property.name, value)
}
```

需要注意的是，即使是如此简单的函数，也可采取这样的创新方式使用公共对象。

Kotlin 支持自定义委托。对此，存在多个库提供了新的属性委托机制，并可在 Android 中用于不同的功能；另外，在 Android 中，属性委托应用方式也多种多样。下面将对相关

示例加以考察，读者将会从中发现该特征的有效性。

8.2.3　自定义委托

之前讨论的全部委托均来自标准库，但读者也可轻松地实现自己的属性委托。如前所述，若将某个类设置为委托，需要提供 getValue 和 setValue 函数，且需要包含实际签名，但无须对类进行扩展或实现接口。当对象用作委托时，并不需要调整其内部实现，其原因在于，可将 getValue 和 setValue 定义为扩展函数。然而，若创建自定义类以作为委托时，接口机制可能十分有用，原因如下：

- 将定义函数结构，因而可在 Android Studio 中生成相应的方法。
- 若正在创建库，则须将委托类指定为私有或内部类，以防止外界的误操作。该情形曾出现于 notNull 部分中，其中，类 NotNullVar 定义为私有类，并用作 ReadWriteProperty<Any?, T>接口。

提供了全功能以支持委托类的接口分别为 ReadOnlyProperty（针对只读属性）和 ReadWriteProperty（针对读写属性），二者十分有用，其定义如下所示：

```
public interface ReadOnlyProperty<in R, out T> {
    public operator fun getValue(thisRef: R,
        property: KProperty<*>): T
}

public interface ReadWriteProperty<in R, T> {
    public operator fun getValue(thisRef: R,
        property: KProperty<*>): T
    public operator fun setValue(thisRef: R,
        property: KProperty<*>, value: T)
}
```

其中，参数值之前已有所讨论，下面再次对其进行回顾，其中包括：

- thisRef 表示对象的引用，其中使用了委托，其类型定义了所用委托的上下文。
- property 表示为引用，包含了与委托属性相关的数据，也就是说，与对应属性相关的全部信息，例如名称和类型。
- value 表示为设置的新值。



thisRef 和 property 参数并未使用下列委托：Lazy、Observable 和 Vetoable。Map、MutableMap 和 notNull 则使用了属性，获取针对键的属性名。但这一类参数可用于不同的场合。

下面考察一些简单有效的自定义属性委托示例，之前曾探讨了针对只读属性的延迟属性委托。然而，某些时候，还需要使用到可变的延迟属性。如果希望在初始化之前请求该值，则应从初始器指定该值并返回该值。在其他场合下，还可用作常规的可变属性，如下所示：

```
fun <T> mutableLazy(initializer: () -> T): ReadWriteProperty<Any?, T> =
MutableLazy<T>(initializer)

private class MutableLazy<T>(val initializer: () -> T) :
ReadWriteProperty<Any?, T> {

   private var value: T? = null
   private var initialized = false

   override fun getValue(thisRef: Any?, property: KProperty<*>): T {
       synchronized(this) {
           if (!initialized) {
               value = initializer()
           }
           return value as T
       }
   }

   override fun setValue(thisRef: Any?,
       property: KProperty<*>, value: T) {
       synchronized(this) {
           this.value = value
           initialized = true
       }
   }
}
```

对于注释 1，委托隐藏于接口之后并通过函数提供服务，因此，可调整 MutableLazy 的实现，且无须担心是否会影响到对其加以使用的代码。

在注释 2 中，实现了 ReadWriteProperty。虽然这是一个可选项，但却十分有用——此处定义了读写属性的正确结构。其中，第一种类型为 Any?，表示可在任意环境下使用该属性委托；第二种类型则为泛型。注意，关于该类型，并不存在限制条件，因而可以是可空类型。

针对注释 3，属性值存储于 value 属性中，并通过初始化后的属性体现。采用该方式的原因在于，支持 T 为可空类型。随后，该值中的 null 表示尚未初始化，或者等于 null。

对于注释 4，无须使用 operator 修饰符却已经用于接口中。

对于注释 5，getValue 在设置任意值之前被调用，随后，该值通过初始化器进行设置；对于注释 6，需要将当前值转换为 T—该值可能为非空；同时可利用 null 作为初始值，进而将该值初始化为可空类型。

在 Android 开发中，上述属性委托在不同场合下十分有用。例如，当属性的默认值存储于某个文件中，且需要对其进行读取时（这将是一项繁重的操作），如下所示：

```
var gameMode : GameMode by MutableLazy {
    getDefaultGameMode()
}

var mapConfiguration : MapConfiguration by MutableLazy {
    getSavedMapConfiguration()
}

var screenResolution : ScreenResolution by MutableLazy {
    getOptimalScreenResolutionForDevice()
}
```

当采用上述方式时，如果用户在应用之前设置该属性的自定义值，则无须亲自对其进行计算。另外，第二个自定义属性委托则可定义属性 getter，如下所示：

```
val a: Int get() = 1
val b: String get() = "KOKO"
val c: Int get() = 1 + 100
```

在 Kotlin 1.1 之前，通常需要定义属性类型。为了避免这一行为，可定义下列函数类型的扩展函数（因而还涉及 Lambda 表达式），如下所示：

```
inline operator fun <R> (() -> R).getValue(
    thisRef: Any?,
    property: KProperty<*>
): R = invoke()
```

随后可采用类似的方式定义属性，如下所示：

```
val a by { 1 }
val b by { "KOKO" }
val c by { 1 + 100 }
```

由于上述方式会导致性能降低，因而这里不建议使用，但这也体现了委托属性的一种可能性。这一类小型扩展函数使得函数类型转为属性委托，并在编译后生成了更加简单的 Kotlin 代码（注意，扩展函数标记为 inline，因而其调用通过函数体替换），如下所示：

```
private val `a$delegate` = { 1 }
val a: Int get() = `a$delegate`()
private val `b$delegate` = { "KOKO" }
val b: String get() = `b$delegate`()
private val `c$delegate` = { 1 + 100 }
val c: Int get() = `c$delegate`()
```

下面将考察实际项目中的某些自定义委托，该机制将随着具体问题一同被提出。

1. 视图绑定

当在项目中采用模型-视图-表示器（MVP）时，需要通过表示器在视图中进行全部修改。因此，可在视图上强制创建多个函数，如下所示：

```
override fun getName(): String {
    return nameView.text.toString()
}

override fun setName(name: String) {
    nameView.text = name
}
```

另外，还可在下列接口中定义函数：

```
interface MainView {
    fun getName(): String
    fun setName(name: String)
}
```

通过属性绑定机制，可简化前述代码，并减少 setter/getter 方法的需求。对此，可针对视图元素绑定对应属性，如下所示：

```
override var name: String by bindToTex(R.id.textView)
```

对应接口如下所示：

```
interface MainView {
    var name: String
}
```

上述代码更加简洁且易于维护。注意，此处通过参数提供了元素 ID。下列代码实现了一个简单的类：

```
fun Activity.bindToText(
    @IdRes viewId: Int ) = object :
    ReadWriteProperty<Any?, String> {
```

```
    val textView by lazy { findViewById<TextView>(viewId) }

    override fun getValue(thisRef: Any?,
        property: KProperty<*>): String {
        return textView.text.toString()
    }

    override fun setValue(thisRef: Any?,
        property: KProperty<*>, value: String) {
        textView.text = value
    }
}
```

针对不同的视图属性以及不同的上下文（Fragment，Service），可创建类似的绑定机制。另一个较为有效的工具是可见性的绑定，也就是说，将某个逻辑属性（对应类型为 Boolean）绑定至 view 元素的可见性上，如下所示：

```
fun Activity.bindToVisibility(
   @IdRes viewId: Int ) = object :
   ReadWriteProperty<Any?, Boolean> {

   val view by lazy { findViewById(viewId) }

   override fun getValue(thisRef: Any?,
       property: KProperty<*>): Boolean {
       return view.visibility == View.VISIBLE
   }

   override fun setValue(thisRef: Any?,
       property: KProperty<*>, value: Boolean) {
       view.visibility = if(value) View.VISIBLE else View.GONE
   }
}
```

上述实现在 Java 中难以完成。类似的绑定行为还可针对其他 View 元素创建，在 MVP 的基础上可实现简单、明了的操作过程。上述代码片段仅为一个简单示例，读者可访问 Kotlin-AndroidViewBindings 库，并查看更多的实现方案（对应网址为 https://github.com/ MarcinMoskala/ KotlinAndroidViewBindings）。

2. 优先绑定

对于更复杂的示例，可在 SharedPreferences 的帮助下予以实现。针对这一问题，Kotlin 提供了更好的解决方法。最终，可将保存于 SharedPreferences 中的数值视作 Shared Preferences

对象的属性。具体应用如下所示：

```
preferences.canEatPie = true
if(preferences.canEatPie) {
    // Code
}
```

下列代码显示了扩展属性定义：

```
var SharedPreferences.canEatPie:
Boolean by bindToPreferenceField(true) // 1
var SharedPreferences.allPieInTheWorld:
Long by bindToPreferenceField(0,"AllPieKey") //2
```

对于注释 1，表示为类型 Boolean 属性。若某个属性非空，则需要在函数的第一个参数中提供默认值。对于注释 2，属性可包含所提供的自定义键，这在实际项目中十分有用，进而可对该键进行操控（例如，在属性重命名时防止意外修改）。

下面通过深入研究非空属性以了解优先绑定的工作方式。对此，首先分析提供方函数。需要注意的是，属性类型由数值与 SharedPreferences 间的获取方式确定（由于存在不同的函数，例如 getString、getInt 等），因而所需类的类型应作为内联函数的 reified 类型，或者通过当前参数这一方式予以提供。对应的委托提供方函数如下所示：

```
inline fun <reified T : Any> bindToPreferenceField(
      default: T?,
      key: String? = null
): ReadWriteProperty<SharedPreferences, T> // 1
    = bindToPreferenceField(T::class, default, key)

fun <T : Any> bindToPreferenceField( // 2
    clazz: KClass<T>,
    default: T?,
    key: String? = null
): ReadWriteProperty<SharedPreferences, T>
      = PreferenceFieldBinder(clazz, default, key) // 1
```

对于注释 1，两个函数均返回基于 ReadWriteProperty<SharedPreferences, T>接口的对象。注意，此处的上下文被设置为 SharedPreferences，因而仅可于此处或 SharedPreferences 扩展内使用。之所以定义该函数，其原因在于：参数类型无法被重定义，且需要作为常规参数提供类型。对于注释 2，bindToPreferenceField 函数不可定义为 private 或 internal，因为内联函数仅可使用包含相同或较少限制的修饰符的函数。

最后考察委托类 PreferenceFieldDelegate，如下所示：

```
internal open class PreferenceFieldDelegate<T : Any>(
      private val clazz: KClass<T>,
      private val default: T?,
      private val key: String?
) : ReadWriteProperty<SharedPreferences, T> {

  override operator fun getValue(thisRef: SharedPreferences,
  property: KProperty<*>): T
    = thisRef.getLong(getValue<T>(clazz, default, getKey(property))

  override fun setValue(thisRef: SharedPreferences,
  property: KProperty<*>, value: T) {
     thisRef.edit().apply
     { putValue(clazz, value, getKey(property)) }.apply()
  }

  private fun getKey(property: KProperty<*>) =
  key ?: "${property.name}Key"
}
```

目前，我们已了解了 thisRef 的使用方式，其类型为 SharedPreferences，并可以此获取或设置全部值。取决于属性类型，用于获取和保存数值的函数定义如下所示：

```
internal fun SharedPreferences.Editor.putValue(clazz: KClass<*>, value:
Any, key: String) {
   when (clazz.simpleName) {
       "Long" -> putLong(key, value as Long)
       "Int" -> putInt(key, value as Int)
       "String" -> putString(key, value as String?)
       "Boolean" -> putBoolean(key, value as Boolean)
       "Float" -> putFloat(key, value as Float)
       else -> putString(key, value.toJson())
   }
}

internal fun <T: Any> SharedPreferences.getValue(clazz: KClass<*>, default:
T?, key: String): T = when (clazz.simpleName) {
   "Long" -> getLong(key, default as Long)
   "Int" -> getInt(key, default as Int)
   "String" -> getString(key, default as? String)
   "Boolean" -> getBoolean(key, default as Boolean)
   "Float" -> getFloat(key, default as Float)
   else -> getString(key, default?.toJson()).fromJson(clazz)
} as T
```

除此之外，还须定义 toJson 和 fromJson，如下所示：

```
var preferencesGson: Gson = GsonBuilder().create()
internal fun Any.toJson() = preferencesGson.toJson(this)!!
internal fun <T : Any> String.fromJson(clazz: KClass<T>) =
preferencesGson.fromJson(this, clazz.java)
```

根据上述定义，可将附加扩展属性定义至 SharedPreferences，如下所示：

```
var SharedPreferences.canEatPie: Boolean by bindToPreferenceField(true)
```

在第 7 章中曾讨论到，Java 中并不存在此类字段可添加至类中。实际上，扩展属性将编译为 getter 和 setter 函数，并将调用委托至所创建的委托中，如下所示：

```
val 'canEatPie$delegate' = bindToPreferenceField(Boolean::class, true)

fun SharedPreferences.getCanEatPie(): Boolean {
  return 'canEatPie$delegate'.getValue(this,
  SharedPreferences::canEatPie)
}

fun SharedPreferences.setCanEatPie(value: Boolean) {
  'canEatPie$delegate'.setValue(this, SharedPreferences::canEatPie,
   value)
}
```

另外，扩展函数实际上表示为在第一个参数上包含扩展的静态函数，如下所示：

```
val 'canEatPie$delegate' = bindToPreferenceField(Boolean::class, true)

fun getCanEatPie(receiver: SharedPreferences): Boolean {
   return 'canEatPie$delegate'.getValue(receiver,
   SharedPreferences::canEatPie)
}

fun setCanEatPie(receiver: SharedPreferences, value: Boolean) {
   'canEatPie$delegate'.setValue(receiver,
   SharedPreferences::canEatPie, value)
}
```

前述示例足以说明属性委托的工作方式以及使用方式。属性委托广泛地应用于 Kotlin 开源项目中，可加速并简化依赖注入（Dependency Injection，例如 Kodein、Injekt 和 TornadoFX）、视图绑定、SharedPreferences 或其他元素（已知内容包括 PreferenceHolder 和 KotlinAndroidViewBindings）的实现，进而设置配置定义上的属性键，例如 Konfig，或者定义数据库列结构，例如 Kwery。除此之外，还存在更为广阔的应用领域等待读者发现。

3. 提供委托

自版本 1.1 之后，Kotlin 提供了 provideDelegate 操作符，并可在类初始化过程中提供委托。provideDelegate 背后的含义是，可根据属性特性（例如名称、类型以及注解等）提供自定义委托。

provideDelegate 操作符返回委托，包含该操作符的全部类型自身无须为委托，对应示例如下所示：

```
class A(val i: Int) {

    operator fun provideDelegate(
        thisRef: Any?,
        prop: KProperty<*>
    ) = object: ReadOnlyProperty<Any?, Int> {

        override fun getValue(
            thisRef: Any?,
            property: KProperty<*>
        ) = i
    }
}

val a by A(1)
```

在该示例中，A 用作委托，但并未实现为 getValue 或 setValue 函数，其原因在于，此处定义了 provideDelegate 操作符，并返回所用的委托，而非 A。相应地，属性委托将编译为下列代码：

```
private val a$delegate = A().provideDelegate(this, this::prop)
val a: Int
get() = a1$delegate.getValue(this, this::prop)
```

具体应用则位于 Kotlin 的 ActivityStarter 库中（对应网址为 https://github. com/ Marcin Moskala/ActivityStarter）。其中，操作参数通过注解加以定义，但可采用属性委托简化 Kotlin 应用，并支持属性定义为只读或非 lateinit，如下所示：

```
@get:Arg(optional = true) val name: String by argExtra(defaultName)
@get:Arg(optional = true) val id: Int by argExtra(defaultId)
@get:Arg val grade: Char by argExtra()
@get:Arg val passing: Boolean by argExtra()
```

当然，一些限制条件依然不可或缺，其中包括：

- 当使用 argExtra 时，须注解属性 getter。
- 若参数可选，且类型不可空，则须指定默认值。

当检测此类需求条件时，须引用属性以获取 getter 注解。在 argExtra 函数中，则无法包含此类引用，但可将其实现于 provideDelegate 中，如下所示：

```
fun <T> Activity.argExtra(default: T? = null) =
ArgValueDelegateProvider(default)
fun <T> Fragment.argExtra(default: T? = null) =
ArgValueDelegateProvider(default)
fun <T> android.support.v4.app.Fragment.argExtra(default: T? = null) =
       ValueDelegateProvider(default)

class ArgValueDelegateProvider<T>(val default: T? = null) {
    operator fun provideDelegate(
        thisRef: Any?,
        prop: KProperty<*>
    ): ReadWriteProperty<Any, T> {
        val annotation = prop.getter.findAnnotation<Arg>()
        when {
            annotation == null ->
            throw Error(ErrorMessages.noAnnotation)
            annotation.optional && !prop.returnType.isMarkedNullable &&
            default == null ->
            throw Error(ErrorMessages.optionalValueNeeded)
        }
        return ArgValueDelegate(default)
    }
}

internal object ErrorMessages {
    const val noAnnotation =
     "Element getter must be annotated with Arg"
    const val optionalValueNeeded =
    "Arguments that are optional and have notnullabletype must have defaut
        value specified"
}
```

若条件未被满足，此类委托将抛出相应的错误，如下所示：

```
val a: A? by ArgValueDelegateProvider()
// Throws error during initialization: Element getter must be
// annotated with Arg
```

```
@get:Arg(optional = true) val a: A by ArgValueDelegateProvider()
//throws error during initialization: Arguments that are optional
//and have not-nullable type must have default value specified.
```

通过这一方式，不被接受的参数定义将在对象初始化过程中抛出相应的错误，而不是以非确定方式中断应用程序。

8.3 本章小结

本章讨论了类委托、属性委托及其应用方式，进而消除冗余代码。其中，可将委托定义为一个对象，并从所委托的其他对象或属性中调用。此过程中还涉及设计模式内容，与类委托关系紧密的设计模式包括委托设计模式和装饰器设计模式。

如前所述，委托模式可视作继承的替代方案；而装饰器模式则可向实现了同一接口的不同类添加功能项。本章介绍了属性委托的工作方式，以及 Kotlin 标准库属性委托，包括 notNull、lazy、observable、vetoable、Map 及其应用方式。此外，本章还探讨了自定义属性委托的生成方式以及应用示例。

需要注意的是，除了理解不同特性及其应用之外，读者还须明晰其整合方式，进而构建大型应用程序。第 9 章将编写一个示例应用程序，并解释各种 Kotlin 特性之间的整合方式。

第 9 章　Marvel Gallery 项目实战

前述章节讨论了 Kotlin 中的大部分重要特性，进而简化 Android 开发并提升效率。但相关内容稍显零散，因而读者难以理解整体项目的实现过程。因此，本章将利用 Kotlin 语言编写一个项目应用程序。

这里的问题是，究竟选择哪一种应用程序予以实现？该程序应具备短小、简单等特征，同时还应兼顾大多数 Kotlin 特性。除此之外，还应减少库的使用量——本书集中讨论 Kotlin 中的 Android 开发，而非 Android 库。同时，应用程序应避免采用手动方式实现图像元素，该过程通常较为复杂，从 Kotlin 角度来看，这并未获得应有的收益。

最终，本章将尝试编写一个 Marvel Gallery 应用程序，该 App 可用于搜索我们喜爱的漫威漫画角色，并显示其详细内容。其中，全部数据均由 Marvel 网站并通过其 API 提供。

9.1　Marvel Gallery 应用程序

Marvel Gallery 应用程序须涵盖下列用例：

- 在启动应用程序后，用户应可看到角色图片库。
- 在启动应用程序后，用户可通过名称搜索角色。
- 当用户单击角色图像时，应显示资料信息，其中包含角色名称、照片、描述以及访问数量。

相应地，存在 3 个用例可描述应用程序的主功能项，稍后将对此逐一予以实现。完整的应用程序内容位于 GitHub 上，对应网址为 https://github.com/MarcinMoskala/MarvelGallery。

为了初步理解构建内容，图 9.1 显示了应用程序最终版本的截图。

9.1.1　如何阅读本章内容

本章展示了构建应用程序所需的全部步骤和代码，以及应用程序开发过程中的各项步骤。在此过程中，读者应关注开发流程，并理解代码功能。相应地，读者无须理解相关布局以及单元测试的定义。对此，读者仅须了解其功能即可。另外，强调应用程序结构以及

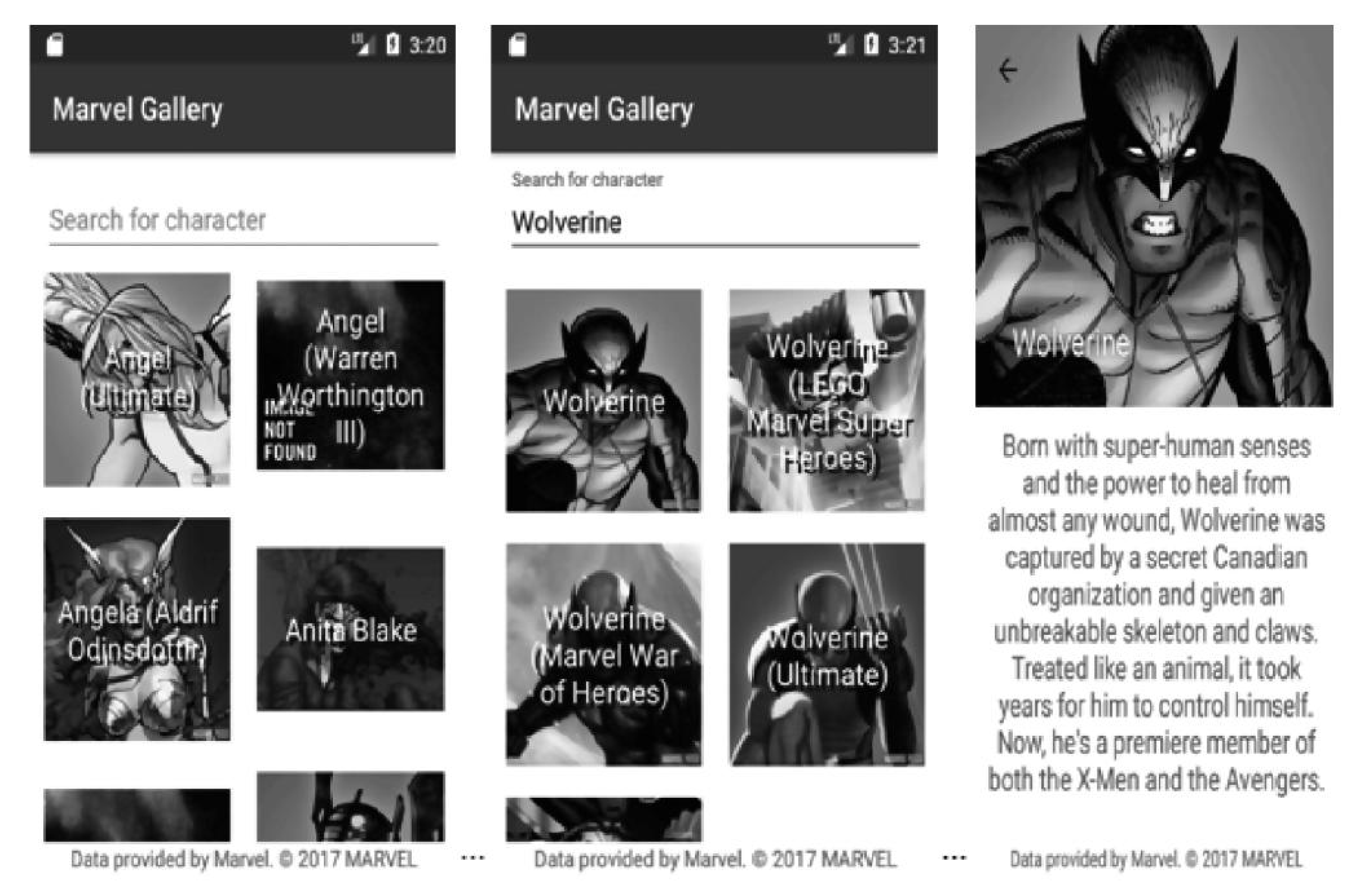


图 9.1

Kotlin 解决方案可使最终代码更加简洁。其中，大多数方案已在前述章节中有所提及，本章仅对其进行简要描述。本章的价值主要体现在：在真实应用程序中阐述具体应用。

读者可下载 GitHub 上的代码，对应网址为 https://github.com/MarcinMoskala/MarvelGallery。

在 GitHub 上，读者可查看最终代码并下载；或者可使用 Git 命令将其克隆至自己的计算机设备上，如下所示：

```
git clone git@github.com:MarcinMoskala/MarvelGallery.git
```

应用程序还包含了采用 Espresso 编写的 UI 测试，但并未出现于本章讨论中，以使内容相对简洁。

本章各部分内容在当前项目中包含对应的 Git 分支，以方便读者查看代码，如图 9.2 所示。

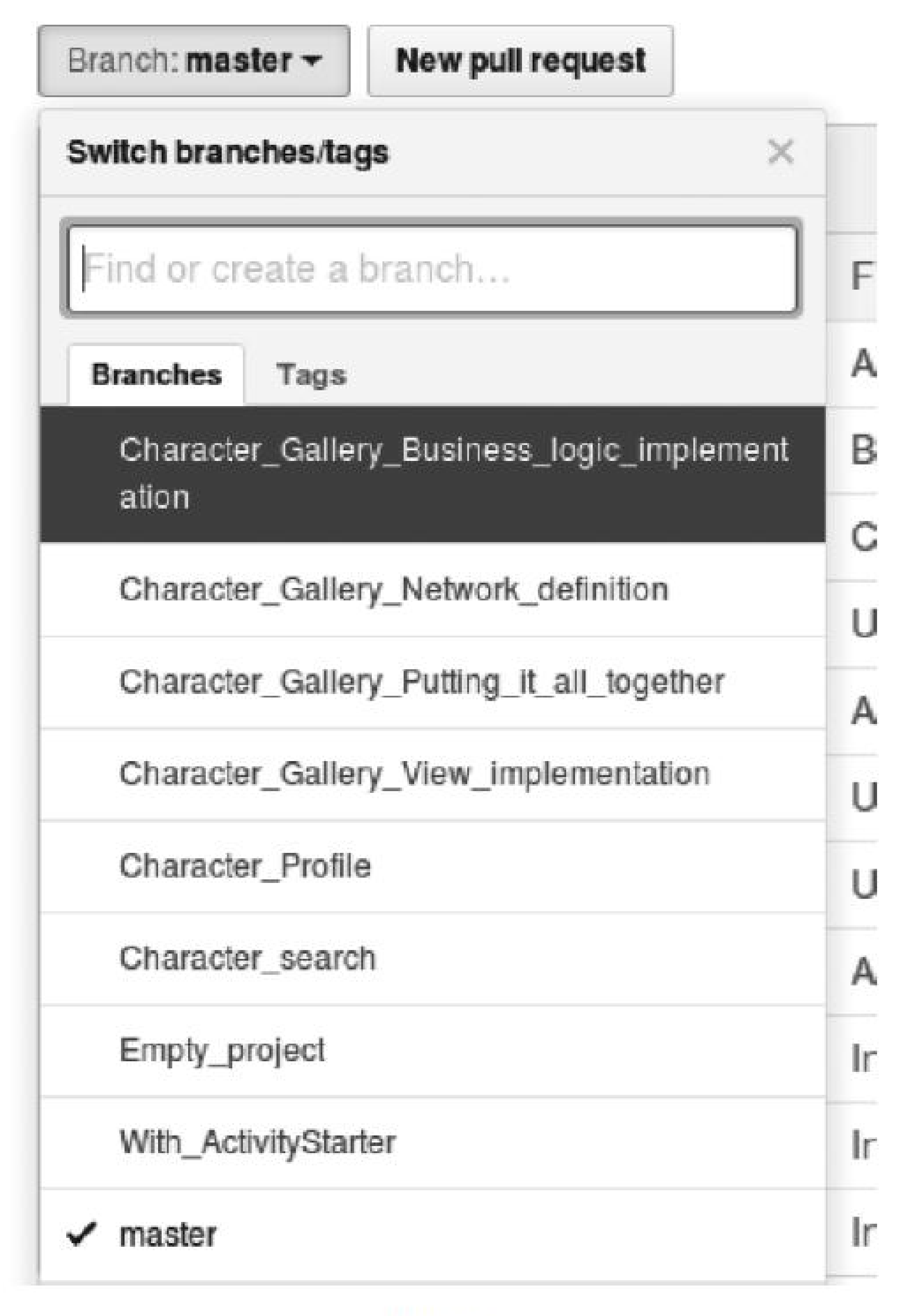


图 9.2

另外，从本地角度来看，在存储库克隆完毕后，可通过 Git 命令检测对应的分支，如下所示：

```
git checkout Character_search
```

如果读者拥有本书的电子版，则可通过复制、粘贴代码生成当前应用程序，但需要将文件置于与当前数据包对应的文件夹内。通过这一方式，可在项目中生成更加清晰的结构。

需要注意的是，若将本书代码置于另一个文件夹中，将产生如图 9.3 所示的警告消息。

```
package com.sample.marvelgallery.view
```

图 9.3

用户可将某个文件置于任意文件夹中，这将把文件放置于所定义的数据包对应的路径中，如图 9.4 所示。

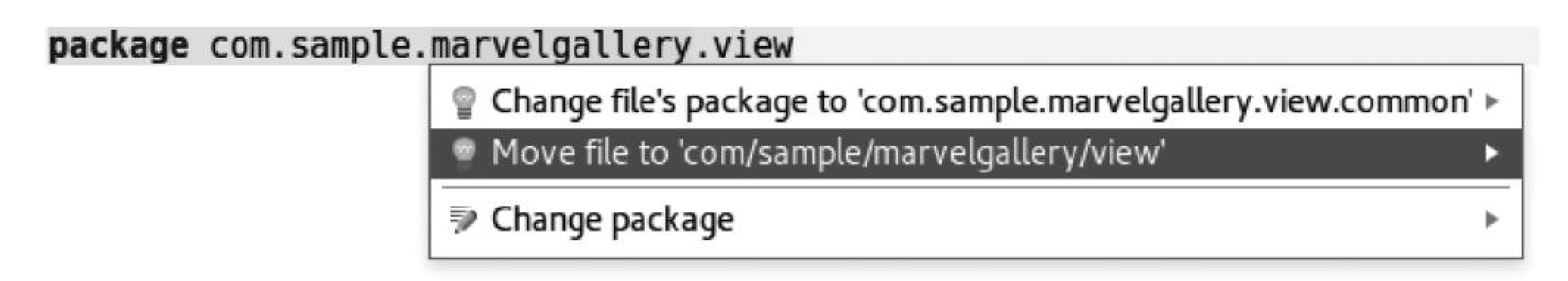


图 9.4

读者可以将此文件置于正确的位置处。

9.1.2　创建空项目

在实现各项功能之前，需要利用 MainActivity 建空的 Kotlin Android 项目，第 1 章曾对此有所讨论，因而此处仅对此予以简要描述；同时，也会逐步考察 Android Studio 3.0 中的各项操作。

（1）设置名称、数据包以及新项目的位置。记住，此处须选中 Include Kotlin support 复选框，如图 9.5 所示。

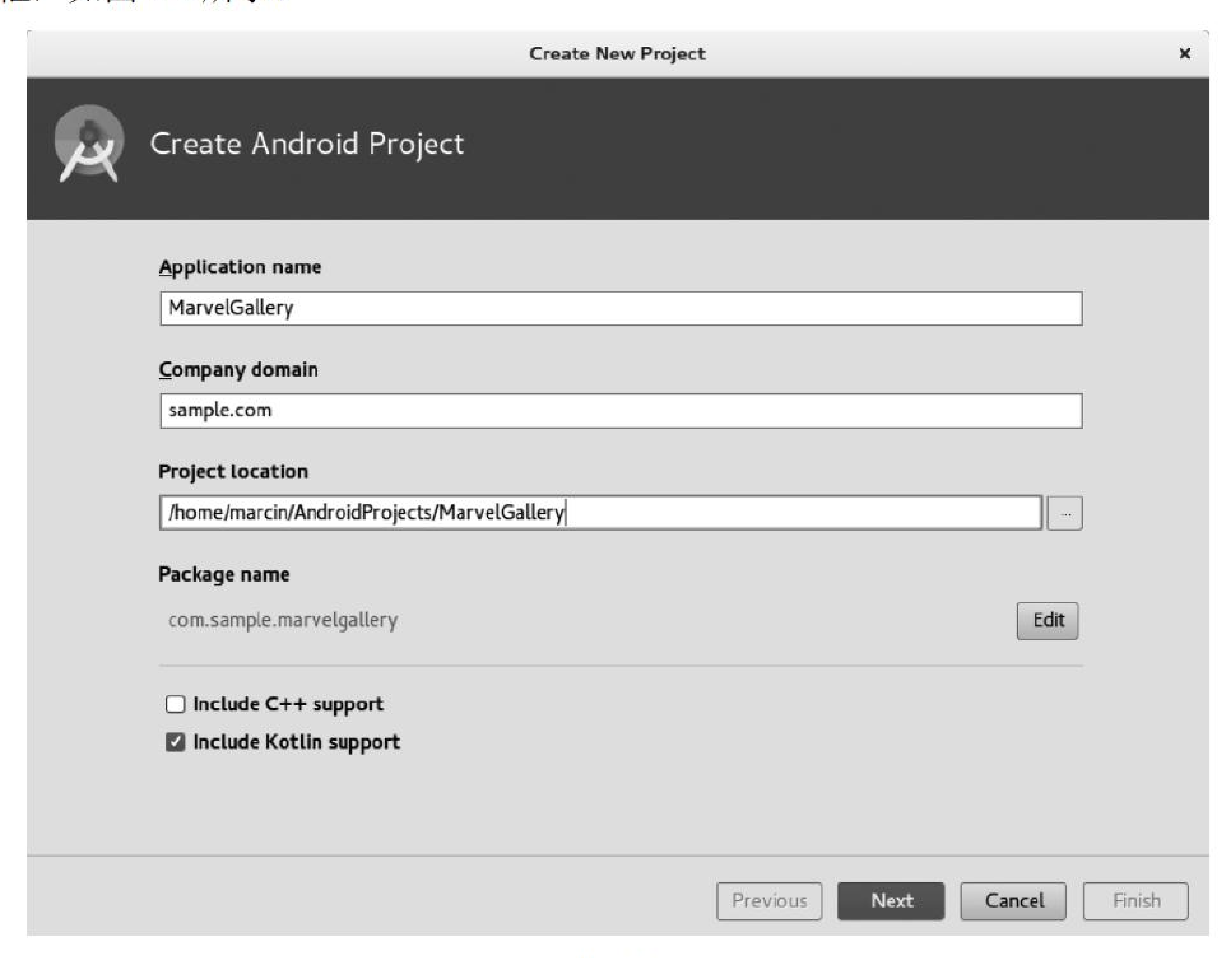


图 9.5

（2）可选择其他 Android 版本。当前示例将此设置为 API 16，如图 9.6 所示。

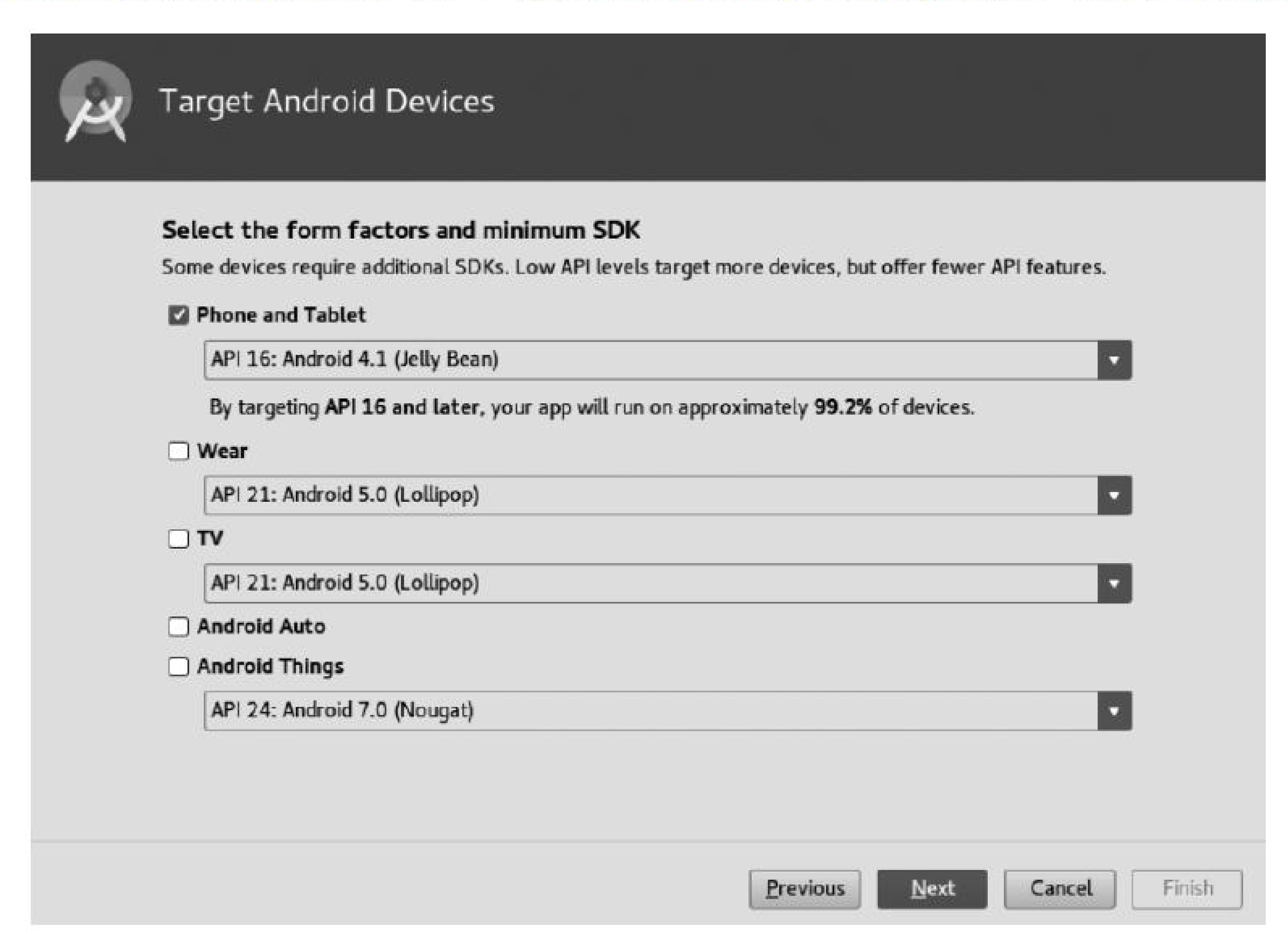


图 9.6

（3）选取模板。当前示例无须使用任何模板，因而应选择 Empty Activity 并启动，如图 9.7 所示。



图 9.7

（4）对新创建的 activity 命名。其中，第一个视图可命名为 MainActivity，如图 9.8 所示。



对于 3.x 之前的 Android Studio，须采用以下步骤：

（1）在当前项目中配置 Kotlin（例如 Ctrl/Cmd + Shift + A 以及 Configure Kotlin in project）。

（2）将所有 Java 类转换为 Kotlin（例如 MainActivity 的 Ctrl/Cmd+Shift+A 和 Convert Java file to Kotlin file）。

随后，即可得到包含空 Activity 的 Kotlin Android 应用程序，如图 9.8 所示。

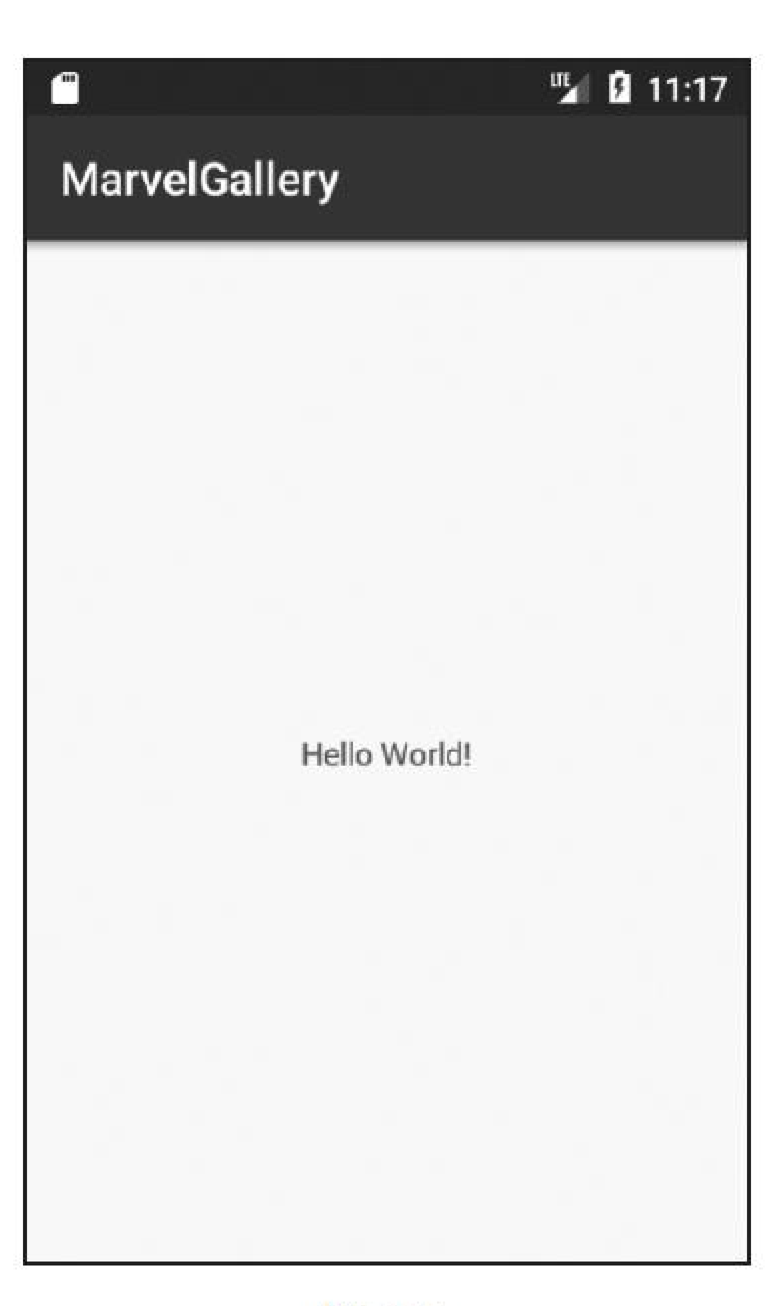


图 9.8

9.1.3　任务图片库

这一部分内容将实现单一用例，也就是说，在启动应用程序后，用户可看到一个任务图片库。

由于需要显示视图，利用 API 实现网络连接以及业务规则，因而该用例相对复杂。因此，可将其分解为多项任务，如下所示：

- 视图实现。
- 利用 API 进行通信。
- 任务显示的业务逻辑实现。
- 整合操作。

下面对此予以逐一实现。

1. 视图实现

下面首先考察视图的实现过程，并定义一个人物角色的外观列表。出于测试目的，这里将定义多个任务并对其加以显示。

对于 MainActivity 布局的实现过程，将采用 RecyclerView 显示元素列表、Recycler View 布局分布于一个单独的依赖项中，须添加 app 模块 build.gradle 文件，如下所示：

```
implementation "com.android.support:recyclerview-
v7:$ android_support_ version"
```

android_support_version 实例表示为一个尚未定义的变量，其后的原因可描述为：对于全部的 Android 支持库，该版本应保持一致；同时，当析取该版本号并作为一个单一变量时，这一操作将易于管理。相应地，对于包含 android_support_version 引用的某一个 Android

支持库，应替换对应的硬编码版本，如下所示：

```
implementation "com.android.support:appcompat
    v7:$android support version"
implementation "com.android.support:design:$android support version"
implementation "com.android.support:support
    v4:$android support version"
implementation "com.android.support:recyclerview
    v7:$android_support_version"
```

同时，还须设置支持库的版本号。对此，一种较好的方法是将其定义于 buildscript 的 build.gradle 项目文件中，并位于 kotlin_version 定义之后，如下所示：

```
ext.kotlin version = '1.1.4-2'
ext.android_support_version = "26.0.1"
```

当前，可查看 MainActivity 布局的实现结果，如图 9.9 所示。

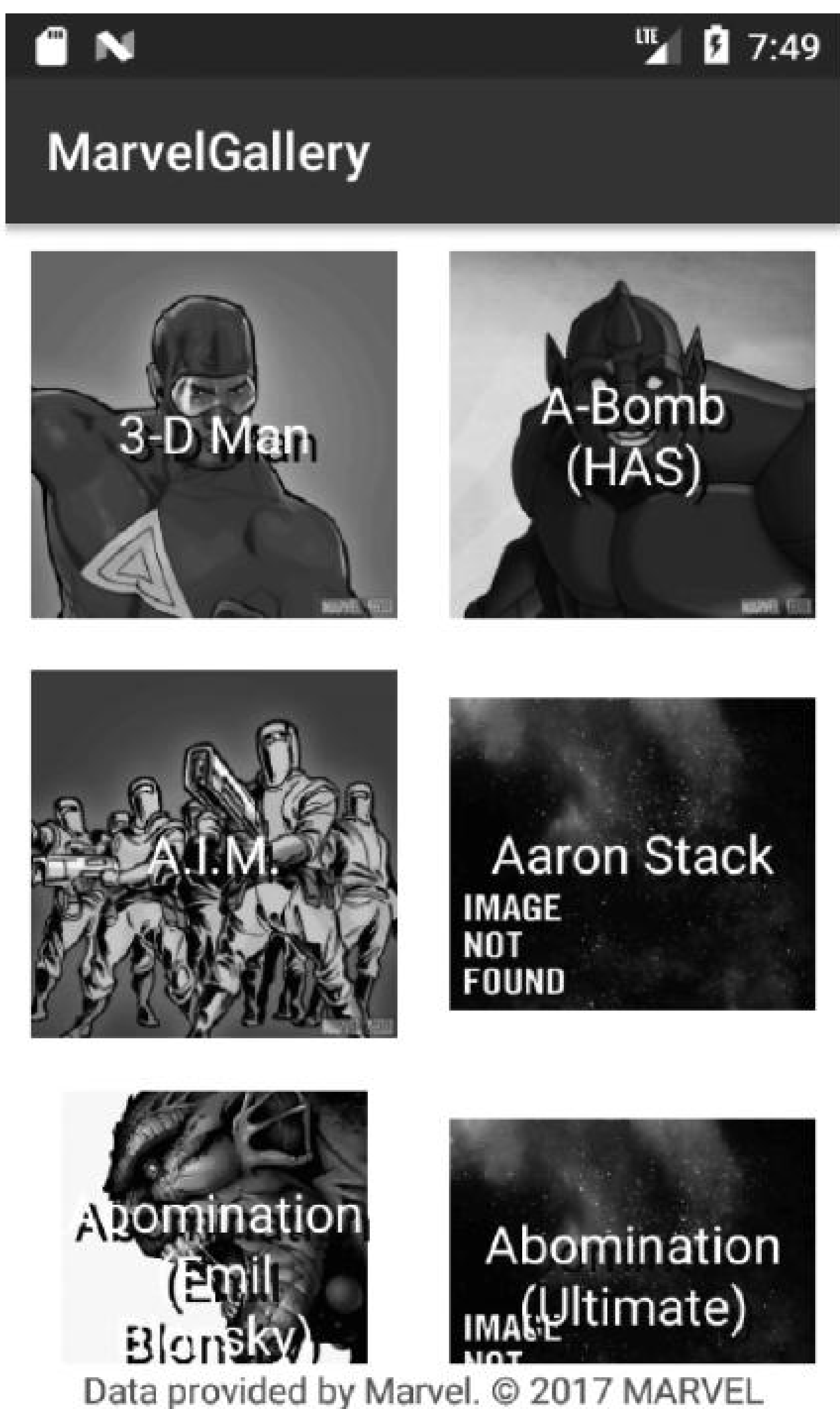


图 9.9

RecyclerView 上的人物元素将封装至 SwipeRefreshLayout，以支持手指滑动更新操作。为了保护漫威的版权，还应显示相关标签，以告知用户数据由 Marvel 所提供。activity_main

（res/layout/activity_main.xml）应利用下列定义予以替换：

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:id="@+id/charactersView"
   android:layout width="match parent"
   android:layout height="match parent"
   android:background="@android:color/white"
   android:fitsSystemWindows="true">

   <android.support.v4.widget.SwipeRefreshLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/swipeRefreshView"
        android:layout width="match parent"
        android:layout height="match parent">

        <android.support.v7.widget.RecyclerView
            android:id="@+id/recyclerView"
            android:layout width="match parent"
            android:layout height="match parent"
            android:scrollbars="vertical" />

    </android.support.v4.widget.SwipeRefreshLayout>

    <TextView
        android:layout width="match parent"
        android:layout height="wrap content"
        android:layout alignParentBottom="true"
        android:background="@android:color/white"
        android:gravity="center"
        android:text="@string/marvel copyright notice" />
</RelativeLayout>
```

这里，需要将版权注意事项添加至字符串中（res/values/strings.xml），如下所示：

```
<string name="marvel copyright notice">
    Data provided by Marvel. . 2017 MARVEL
</string>
```

图 9.10 显示了预览效果。

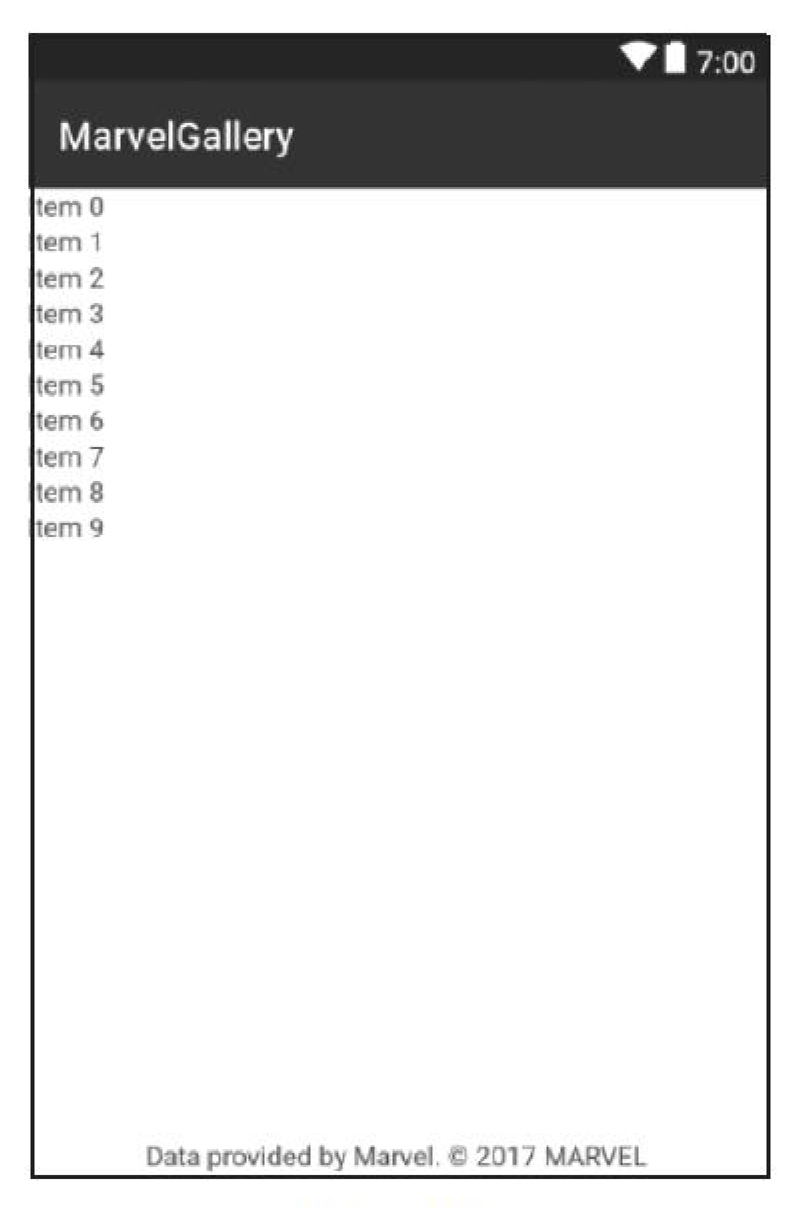


图 9.10

下一个步骤是定义各项视图。其中，每个元素项将显示为正方形。对此，需要定义一个持有正方形形状的视图（将其置于 view/views 中），如下所示：

```
package com.sample.marvelgallery.view.views

import android.util.AttributeSet
import android.widget.FrameLayout
import android.content.Context

class SquareFrameLayout @JvmOverloads constructor( // 1
       context: Context,
       attrs: AttributeSet? = null,
       defStyleAttr: Int = 0
) : FrameLayout(context, attrs, defStyleAttr) {

    override fun onMeasure(widthMeasureSpec: Int,
    heightMeasureSpec: Int) {
        super.onMeasure(widthMeasureSpec, widthMeasureSpec) // 2
    }
}
```

针对注释 1，当使用 JvmOverloads 注解时，可避免调整在 Android 中设置自定义视图的构造函数，具体内容可参考第 4 章；对于注释 2，强制元素的高度与宽度相同。

利用 SquareFrameLayout，可定义图库各项的布局，如图 9.11 所示。



图 9.11

同时，还须定义 ImageView 以显示人物角色图像，以及 TextView 以显示其名称。实际上，虽然 SquareFrameLayout 表示为 FrameLayout（包含固定高度），但其子元素（图像和文本）在默认时彼此重叠。下面将布局添加至 res/layout 的 item_character.xml 文件中，如下所示：

```
// ./res/layout/item character.xml

<com.sample.marvelgallery.view.views.SquareFrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout width="match parent"

   android:layout height="wrap content"
   android:gravity="center horizontal"
   android:orientation="horizontal"
   android:padding="@dimen/element padding">

   <ImageView
       android:id="@+id/imageView"
       android:layout width="match parent"
       android:layout height="match parent"/>

   <TextView
       android:id="@+id/textView"
       android:layout width="match parent"
       android:layout_height="match_parent"
```

```
        android:gravity="center"
        android:paddingLeft="10dp"
        android:paddingRight="10dp"
        android:shadowColor="#111"
        android:shadowDx="5"
        android:shadowDy="5"
        android:shadowRadius="0.01"
        android:textColor="@android:color/white"
        android:textSize="@dimen/standard text size"
        tools:text="Some name" />
</com.sample.marvelgallery.view.views.SquareFrameLayout>
```

注意，此处使用了诸如 element_padding 这一类值（定义于 dimens），下面将其添加至 res/values 中的 dimen.xml 文件中，如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <dimen name="character header height">240dp</dimen>
   <dimen name="standard text size">20sp</dimen>
   <dimen name="character description padding">10dp</dimen>
   <dimen name="element padding">10dp</dimen>
</resources>
```

从中可看到，每个元素都需要显示人物角色的冰川及其图像。因此，角色模型须包含两项属性。下面针对某个角色定义一个简单的模型，如下所示：

```
package com.sample.marvelgallery.model

data class MarvelCharacter(
        val name: String,
        val imageUrl: String
)
```

当采用 RecyclerView 显示元素列表时，需要实现列表 RecyclerView 和数据项适配器。其中，列表适配器用于管理列表中的全部元素；而数据项适配器则是针对单一数据项类型的适配器。相应地，此处仅需要一个数据项适配器，因为当前仅显示单一的数据项类型。一种假设是，未来还可能在列表中加入其他类型的元素，例如漫画或广告。在大多数项目中，往往会存在多个单一列表，对此，一种较好的做法是将公共行为抽取至某个独立抽象类中。

尽管该示例旨在展示 Kotlin 在大型项目中的应用方式，但这里将定义一个抽象列表适配器，并将其命名为 RecyclerListAdapter，以及一个命名为 ItemAdapter 的抽象数据项适配器。其中，ItemAdapter 的定义如下所示：

```
package com.sample.marvelgallery.view.common

import android.support.v7.widget.RecyclerView
import android.support.annotation.LayoutRes
import android.view.View

abstract class ItemAdapter<T : RecyclerView.ViewHolder>
(@LayoutRes open val layoutId: Int) { // 1

   abstract fun onCreateViewHolder(itemView: View): T // 2

   @Suppress("UNCHECKED_CAST") // 1
   fun bindViewHolder(holder: RecyclerView.ViewHolder) {
       (holder as T).onBindViewHolder() // 1
   }

   abstract fun T.onBindViewHolder() // 1, 3
}
```

针对注释 1，须作为类型参数传递一个加载器，并可在其字段上直接进行操作。该加载器创建于 onCreateViewHolder 中，因而可知其类型通常为类型参数 T。因此，可将该加载器转换为 bindViewHolder 上的 T，并将其用作针对 onBindViewHolder 的接收器对象。

对于注释 2，函数用于创建视图加载器。在大多数时候，可表示为一个单一的表达式函数，且仅调用构造函数。

对于注释 3，在 onBindViewHolder 函数中，将设置数据项视图上的全部数值。

RecyclerListAdapter 的定义如下所示：

```
package com.sample.marvelgallery.view.common

import android.support.v7.widget.RecyclerView
import android.view.LayoutInflater
import android.view.ViewGroup

open class RecyclerListAdapter( // 1
       var items List<AnyItemAdapter> = listOf()
) : RecyclerView.Adapter<RecyclerView.ViewHolder>() {

   override final fun getItemCount() = items.size // 4

   override final fun getItemViewType(position: Int) =
       items[position].layoutId // 3, 4
```

```
    override final fun onCreateViewHolder(parent: ViewGroup,
        layoutId: Int): RecyclerView.ViewHolder { // 4
    val itemView = LayoutInflater.from(parent.context)
        .inflate(layoutId, parent, false)
        return items.first
        { it.layoutId == layoutId }.onCreateViewHolder(itemView) // 3
    }

    override final fun onBindViewHolder
    (holder: RecyclerView.ViewHolder, position: Int) { // 4
        items[position].bindViewHolder(holder)
    }
}

typealias AnyItemAdapter = ItemAdapter
    <out RecyclerView.ViewHolder> // 5
```

针对注释 1，此处未采用抽象机制，其原因在于，无须使用任何子元素即可初始化并加以使用。针对不同的列表，可定义子元素，进而设置自定义方法；对于注释 2，将数据项保存于列表中；对于注释 3，将采用布局区分数据项类型。据此，将无法使用同一列表上包含相同布局的两个数据项适配器，但该方案可使问题得以简化；对于注释 4，对应方法覆写了 RecyclerView.Adapter 中的方法，但也使用了 final 修饰符限制子元素中的覆写操作。扩展了 RecyclerListAdapter 的全部列表应对数据项进行操作；对于注释 5，定义了类型别名以简化任意的 ItemAdapter 定义。

当采用上述定义时，可定义 MainListAdapter（针对人物列表的适配器）以及 CharacterItemAdapter（针对列表上数据项的列表）。MainListAdapter 的定义如下所示：

```
package com.sample.marvelgallery.view.main

import com.sample.marvelgallery.view.common.AnyItemAdapter
import com.sample.marvelgallery.view.common.RecyclerListAdapter

class MainListAdapter(items: List<AnyItemAdapter>) :
RecyclerListAdapter(items)
```

在该项目中，无须使用定义于 MainListAdapter 中的特定方法，但为了展示其定义的简单性，下列代码展示了包含附加方法（add 和 delete）的 MainListAdapter。

```
class MainListAdapter(items: List<AnyItemAdapter>) :
RecyclerListAdapter(items) {

    fun add(itemAdapter: AnyItemAdapter) {
```

```
        items += itemAdapter)
        val index = items.indexOf(itemAdapter)
        if (index == -1) return
        notifyItemInserted(index)
    }

    fun delete(itemAdapter: AnyItemAdapter) {
        val index = items.indexOf(itemAdapter)
        if (index == -1) return
        items -= itemAdapter
        notifyItemRemoved(index)
    }
}
```

CharacterItemAdapter 的定义如下所示：

```
package com.sample.marvelgallery.view.main

import android.support.v7.widget.RecyclerView
import android.view.View
import android.widget.ImageView
import android.widget.TextView
import com.sample.marvelgallery.R
import com.sample.marvelgallery.model.MarvelCharacter
import com.sample.marvelgallery.view.common.ItemAdapter
import com.sample.marvelgallery.view.common.bindView
import com.sample.marvelgallery.view.common.loadImage

class CharacterItemAdapter(
       val character: MarvelCharacter // 1
):ItemAdapter<CharacterItemAdapter.ViewHolder>(R.layout.item character) {

   override fun onCreateViewHolder(itemView: View) = ViewHolder(itemView)

    override fun ViewHolder.onBindViewHolder() { // 2
        textView.text = character.name
        imageView.loadImage(character.imageUrl) // 3
    }

    class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView)
    {
        val textView by bindView<TextView>(R.id.textView) // 4
        val imageView by bindView<ImageView>(R.id.imageView) // 4
    }
}
```

针对注释 1，MarvelCharacter 通过构造函数传递；针对注释 2，onBindViewHolder 用于设置视图，并定义为 ItemAdapter 中的抽象成员扩展函数。据此，可在其函数体中显式地使用 textView 和 imageView；对于注释 3，函数 loadImage 尚未定义，稍后将其定义为扩展函数；对于注释 4，在视图加载器中，通过 bindView 函数（稍后将对其加以定义），将属性绑定至视图元素上。

在内部，可使用 loadImage 和 bindView 函数（尚未定义）。另外，bindView 表示为针对 RecyclerView.ViewHolder 的顶级扩展函数，并提供了延迟委托（并通过其 ID 提供一个视图），如下所示：

```
// ViewExt.kt
package com.sample.marvelgallery.view.common

import android.support.v7.widget.RecyclerView
import android.view.View

fun <T : View> RecyclerView.ViewHolder.bindView(viewId: Int)
      = lazy { itemView.findViewById<T>(viewId) }
```

除此之外，还需定义 loadImage 扩展函数，以帮助我们从 URL 下载一幅图像，并将其置入 ImageView。对此，可使用两个典型的库，即 Picasso 和 Glide。此处选择了 Glide。对此，需要在 build.gradle 中添加依赖项，如下所示：

```
implementation "com.android.support:recyclerview
v7:$ android support version"
implementation "com.github.bumptech.glide:glide:$glide_version"
```

确定项目 build.gradle 中的当前版本，如下所示：

```
ext.android support version = "26.0.0"
ext.glide_version = "3.8.0"
```

添加权限，并在 AndroidManifest 中使用互联网，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.sample.marvelgallery">
   <uses-permission android:name="android.permission.INTERNET" />
   <application
...
```

最后，可针对 ImageView 类定义 loadImage 扩展函数，如下所示：

```
// ViewExt.kt
package com.sample.marvelgallery.view.common
```

```
import android.support.v7.widget.RecyclerView
import android.view.View
import android.widget.ImageView
import com.bumptech.glide.Glide

fun <T : View> RecyclerView.ViewHolder.bindView(viewId: Int)
       = lazy { itemView.findViewById<T>(viewId) }

fun ImageView.loadImage(photoUrl: String) {
   Glide.with(context)
           .load(photoUrl)
           .into(this)
}
```

至此，可定义 Activity 显示该列表，并使用 Kotlin Android 扩展插件，用于简化代码中视图元素的访问行为。具体过程十分简单，即在 build.gradle 模块中加入 kotlin-android-extensions 插件，如下所示：

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
```

And we have some view defined in layout:

```
<TextView
  android:id="@+id/nameView"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content" />
```

随后，可在 Activity 中导入指向该视图的引用，如下所示：

```
import kotlinx.android.synthetic.main.activity_main.*
```

接下来，通过名称即可直接访问 View 元素，且无须使用 findViewById 方法或定义注解，如下所示：

```
nameView.text = "Some name"
```

在当前项目的全部 Activity 中，均会使用 Kotlin Android 扩展。下面定义 Main Activity，以显示包含图像的人物角色列表，如下所示：

```
package com.sample.marvelgallery.view.main

import android.os.Bundle
import android.support.v7.app.AppCompatActivity
```

```
import android.support.v7.widget.GridLayoutManager
import android.view.Window
import com.sample.marvelgallery.R
import com.sample.marvelgallery.model.MarvelCharacter
import kotlinx.android.synthetic.main.activity main.*

class MainActivity : AppCompatActivity() {

   private val characters = listOf( // 1
       MarvelCharacter(name = "3-D Man", imageUrl =
"http://i.annihil.us/u/prod/marvel/i/mg/c/e0/535fecbbb9784.jpg"),
        MarvelCharacter(name = "Abomination (Emil Blonsky)", imageUrl =
"http://i.annihil.us/u/prod/marvel/i/mg/9/50/4ce18691cbf04.jpg")
   )

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       requestWindowFeature(Window.FEATURE NO TITLE) // 2
       setContentView(R.layout.activity main)
       recyclerView.layoutManager = GridLayoutManager(this, 2) // 3
       val categoryItemAdapters = characters
       .map(::CharacterItemAdapter) // 4
       recyclerView.adapter = MainListAdapter(categoryItemAdapters)
   }
}
```

对于注释 1，定义了一个可供显示的临时人物角色列表；对于注释 2，由于并未显示标题，此处使用了当前窗口特性；对于注释 3，使用 GridLayoutManager 作为 RecyclerView 布局管理器，以实现网格效果；对于注释 4，可通过 CharacterItem Adapter 构造函数引用，并根据角色人物创建数据项适配器。

至此，可编译当前项目，最终效果如图 9.12 所示。



图 9.12

2. 网络定义

截止到目前，所显示的数据均在应用程序内以硬编码方式体现；除此之外，还需要使用到源自 Marvel API 的数据。对此，须定义相应的网络机制，并从服务器处接收数据。这里将使用一个较为流行的 Android 库 Retrofit，进而简化网络操作；同时，还将用到另一个库 RxJava，用于实

现响应式编程。对于这两个库，将仅使用基本的功能，并尽可能简化应用过程。在应用过程中，需要在 build.gradle 模块中添加下列依赖项：

```
dependencies {
   implementation "org.jetbrains.kotlin:kotlin-stdlib-jre7:
   $kotlin version"
   implementation "com.android.support:appcompat-v7:
   $android support version"
   implementation "com.android.support:recyclerview-v7:
   $android support version"
   implementation "com.github.bumptech.glide:glide:$glide version"

   // RxJava
   implementation "io.reactivex.rxjava2:rxjava:$rxjava version"

   // RxAndroid
   implementation "io.reactivex.rxjava2:rxandroid:$rxandroid version"

   // Retrofit
   implementation(["com.squareup.retrofit2:retrofit:$retrofit version",
                   "com.squareup.retrofit2:adapterrxjava2:$retrofit v
                   ersion",
                   "com.squareup.retrofit2:convertergson:$retrofit ve
                   rsion",
                   "com.squareup.okhttp3:okhttp:$okhttp version",
                   "com.squareup.okhttp3:logginginterceptor:$okhttp v
                   ersion"])

  testImplementation 'junit:junit:4.12'
  androidTestImplementation
  'com.android.support.test:runner:1.0.0'
  androidTestImplementation
  'com.android.support.test.espresso:espresso-core:3.0.0'
}
```

项目 build.gradle 中另一个版本定义如下所示：

```
ext.kotlin version = '1.1.3-2'
ext.android support version = "26.0.0"
ext.glide version = "3.8.0"

ext.retrofit version = '2.2.0'
ext.okhttp version = '3.6.0'
ext.rxjava version = "2.1.2"
ext.rxandroid_version = '2.0.1'
```

我们已经获得定义于 AndroidManifest 中的网络授权，因而无须再次添加。简单的 Retrofit 定义如下所示：

```
val retrofit by lazy { makeRetrofit() } // 1

private fun makeRetrofit(): Retrofit = Retrofit.Builder()
        .baseUrl("http://gateway.marvel.com/v1/public/") // 2
        .build()
```

对于注释 1，可将 retrofit 实例视为延迟顶级属性；而对于注释 2，此处定义了 baseUrl。

除此之外，Retrofit 还包含一些附加需求条件，且需要予以满足。例如，须添加转换器，以实现 Retrofit 和 RxJava 之间的整合应用，并将对象序列化为 JSON。另外，还需要使用到拦截器以提供 Marvel API 所需的数据头和额外的查询操作。鉴于当前示例仅是一类小型应用程序，因而可将全部所需元素定义为顶级函数。Retrofit 的完整定义如下所示：

```
// Retrofit.kt
package com.sample.marvelgallery.data.network.provider

import com.google.gson.Gson
import okhttp3.OkHttpClient
import retrofit2.Retrofit
import retrofit2.adapter.rxjava2.RxJava2CallAdapterFactory
import retrofit2.converter.gson.GsonConverterFactory
import java.util.concurrent.TimeUnit

val retrofit by lazy { makeRetrofit() }

private fun makeRetrofit(): Retrofit = Retrofit.Builder()
        .baseUrl("http://gateway.marvel.com/v1/public/")
        .client(makeHttpClient())
        .addConverterFactory(GsonConverterFactory.create(Gson())) // 1
        .addCallAdapterFactory(RxJava2CallAdapterFactory.create()) // 2
        .build()

private fun makeHttpClient() = OkHttpClient.Builder()
        .connectTimeout(60, TimeUnit.SECONDS) // 3
        .readTimeout(60, TimeUnit.SECONDS) // 4
        .addInterceptor(makeHeadersInterceptor()) // 5
        .addInterceptor(makeAddSecurityQueryInterceptor()) // 6
        .addInterceptor(makeLoggingInterceptor()) // 7
        .build()
```

对于注释 1，添加转换器以支持对象的 JSON 序列化操作和反序列化操作（使用 GSON 库）；

对于注释 2，针对网络请求的返回值，添加转换器以使 RxJava2 类型（包括 Observable 和 Single）为 Observable；对于注释 3，添加自定义拦截器，并对所有内容加以定义。

下列代码定义了所需的拦截器。其中，makeHeadersInterceptor 用于向各个请求添加标准的数据头。

```
// HeadersInterceptor.kt
package com.sample.marvelgallery.data.network.provider

import okhttp3.Interceptor

fun makeHeadersInterceptor() = Interceptor { chain -> // 1
   chain.proceed(chain.request().newBuilder()
           .addHeader("Accept", "application/json")
           .addHeader("Accept-Language", "en")
           .addHeader("Content-Type", "application/json")
           .build())
}
```

对于注释 1，拦截器表示为 SAM，因而可采用 SAM 构造函数对其加以定义。

当在调试模式下运行当前应用程序时，makeLoggingInterceptor 函数用于显示控制台日志，如下所示：

```
// LoggingInterceptor.kt
package com.sample.marvelgallery.data.network.provider

import com.sample.marvelgallery.BuildConfig
import okhttp3.logging.HttpLoggingInterceptor

fun makeLoggingInterceptor() = HttpLoggingInterceptor().apply {
   level = if (BuildConfig.DEBUG) HttpLoggingInterceptor.Level.BODY
          else HttpLoggingInterceptor.Level.NONE
}
```

makeAddRequiredQueryInterceptor 函数稍显复杂，该函数用于提供 Marvel API 所用的查询参数，进而对用户进行验证。这一类参数须采用 MD5 算法计算哈希值，同时还须使用到源自 Marvel API 的公钥和私钥。在 https://developer.marvel.com/上，每名用户将生成自己的密钥。待操作完毕后，须将其置入 gradle.properties 文件中，如下所示：

```
org.gradle.jvmargs=-Xmx1536m
marvelPublicKey=REPLEACE_WITH_YOUR_PUBLIC_MARVEL_KEY
marvelPrivateKey=REPLEACE_WITH_YOUR_PRIVATE_MARVEL_KEY
```

另外，还须在 defaultConfig 部分的 build.gradle 模块中加入下列定义：

```
defaultConfig {
   applicationId "com.sample.marvelgallery"
   minSdkVersion 16
   targetSdkVersion 26
   versionCode 1
   versionName "1.0"
   testInstrumentationRunner
   "android.support.test.runner.AndroidJUnitRunner"
   buildConfigField("String", "PUBLIC KEY", "\"${marvelPublicKey}\"")
   buildConfigField("String", "PRIVATE KEY", "\"${marvelPrivateKey}\"")
}
```

待项目重新构造完毕后，可通过 BuildConfig.PUBLIC_KEY 和 BuildConfig.PRIVATE_ KEY 访问相关值。通过此类密钥，可生成 Marvel API 所需的查询参数，如下所示：

```
  // QueryInterceptor.kt
  package com.sample.marvelgallery.data.network.provider

  import com.sample.marvelgallery.BuildConfig
  import okhttp3.Interceptor

  fun makeAddSecurityQueryInterceptor() = Interceptor { chain ->
     val originalRequest = chain.request()
     val timeStamp = System.currentTimeMillis()

     // Url customization: add query parameters
     val url = originalRequest.url().newBuilder()
             .addQueryParameter("apikey", BuildConfig.PUBLIC KEY) // 1
             .addQueryParameter("ts", "$timeStamp") // 1
             .addQueryParameter("hash", calculatedMd5(timeStamp.toString()+
  BuildConfig.PRIVATE KEY + BuildConfig.PUBLIC KEY)) // 1
             .build()

  // Request customization: set custom url
  val request = originalRequest
          .newBuilder()
          .url(url)
          .build()

  chain.proceed(request)
}
```

关于注释 1，我们需要提供 3 个额外查询，如下所示。

- apikey：包含了当前的公钥。
- ts：仅包含了设备的时间（以毫秒计），用于改善下一次查询所提供的哈希值的安全性。
- hash：根据时间戳、私钥和公钥（逐一保存在独立的 String 中）计算为 MD5 哈希值。

用于计算 MD5 哈希值的函数定义如下所示：

```
// MD5.kt
package com.sample.marvelgallery.data.network.provider

import java.math.BigInteger
import java.security.MessageDigest

/**
* Calculate MD5 hash for text
* @param timeStamp Current timeStamp
* @return MD5 hash string
*/
fun calculatedMd5(text: String): String {
   val messageDigest = getMd5Digest(text)
   val md5 = BigInteger(1, messageDigest).toString(16)
   return "0" * (32 - md5.length) + md5 // 1
}

private fun getMd5Digest(str: String): ByteArray =
MessageDigest.getInstance("MD5").digest(str.toByteArray())

private operator fun String.times(i: Int) = (1..i).fold("") { acc, -> acc+
this }
```

关于注释 1，若小于 32 位，则使用时间扩展操作符，并利用 0 值填充。

之前曾定义了拦截器，因而可在此定义实际的 API 方法。Marvel API 涵盖了大量的数据模型，并以此体现人物角色、列表等内容。相应地，可将其定义为单独的类，并称之为数据传输对象（DTO）。下列代码定义了所需的对象。

```
package com.sample.marvelgallery.data.network.dto

class DataContainer<T> {
   var results: T? = null
}

package com.sample.marvelgallery.data.network.dto
```

```
class DataWrapper<T> {
    var data: DataContainer<T>? = null
}

package com.sample.marvelgallery.data.network.dto

class ImageDto {

    lateinit var path: String // 1
    lateinit var extension: String // 1

    val completeImagePath: String
        get() = "$path.$extension"
}

package com.sample.marvelgallery.data.network.dto

class CharacterMarvelDto {
    lateinit var name: String // 1
    lateinit var thumbnail: ImageDto // 1

    val imageUrl: String
        get() = thumbnail.completeImagePath
}
```

关于注释 1，针对未提供的数据值，应设置相应的默认值；而强制值可采用 lateinit 作为前缀。

Retrofit 采用反射生成 HTTP 请求（根据接口定义），这也是定义 HTTP 请求接口的实现方式，如下所示：

```
package com.sample.marvelgallery.data.network

import com.sample.marvelgallery.data.network.dto.CharacterMarvelDto
import com.sample.marvelgallery.data.network.dto.DataWrapper
import io.reactivex.Single
import retrofit2.http.GET
import retrofit2.http.Query

interface MarvelApi {

    @GET("characters")
    fun getCharacters(
            @Query("offset") offset: Int?,
```

```
        @Query("limit") limit: Int?
    ): Single<DataWrapper<List<CharacterMarvelDto>>>
}
```

根据这一定义，最终可获得人物角色列表，如下所示：

```
retrofit.create(MarvelApi::class.java) // 1
    .getCharacters(0, 100) // 2
    .subscribe({ /* code */ }) // 3
```

针对注释 1，可使用 retrofit 实例创建某个对象，并根据 MarvelApi 接口定义生成 HTTP 请求；对于注释 2，将创建 observable 并将调用传递至 API；对于注释 3，通过订阅功能，将发送 HTTP 请求，并启动响应监听。第一个参数表示为当前回调，并在成功接收响应时被调用。

上述网络定义可满足要求，但还可采用更优的方式予以实现。目前，最大的问题是需要在 DTO 对象上进行操作，而非自己的数据模型对象。对于映射机制，须定义一个附加层。对此，可采用存储库模式。当实现单元测试时，该模式十分有用，其原因在于：可模拟存储库而不是 API 整体定义。这也是我们期望持有的储存库定义，如下所示：

```
package com.sample.marvelgallery.data

import com.sample.marvelgallery.model.MarvelCharacter
import io.reactivex.Single

interface MarvelRepository {

    fun getAllCharacters(): Single<List<MarvelCharacter>>
}
```

MarvelRepository 的实现过程如下所示：

```
package com.sample.marvelgallery.data

import com.sample.marvelgallery.data.network.MarvelApi
import com.sample.marvelgallery.data.network.provider.retrofit
import com.sample.marvelgallery.model.MarvelCharacter
import io.reactivex.Single

class MarvelRepositoryImpl : MarvelRepository {

    val api = retrofit.create(MarvelApi::class.java)

    override fun getAllCharacters(): Single<List<MarvelCharacter>> =
```

```
api.getCharacters(
            offset = 0,
            limit = elementsOnListLimit
   ).map {
       it.data?.results.orEmpty().map(::MarvelCharacter) // 1
   }

   companion object {
       const val elementsOnListLimit = 50
   }
}
```

对于注释 1，可获得 DTO 元素列表，并通过构造函数引用将其映射至 MarvelCharacter 中。

为了保证正常工作，需要在 MarvelCharacter 中定义额外的构造函数，并将 Character MarvelDto 作为参数，如下所示：

```
package com.sample.marvelgallery.model

import com.sample.marvelgallery.data.network.dto.CharacterMarvelDto

class MarvelCharacter(
        val name: String,
        val imageUrl: String
) {

  constructor(dto: CharacterMarvelDto) : this(
          name = dto.name,
          imageUrl = dto.imageUrl
  )
}
```

相应地，存在不同的方式提供 MarvelRepository 实例。在大多数常见实现中，Marvel Repository 的具体实例将作为参数传递至 Presenter。但是，关于 UI 测试（例如 Espresso 测试），情况又当如何？此处并不打算测试 Marvel API，以及在此基础上的 UI 测试。具体的解决方案可描述为：制定某种机制并在运行期内生成标准实现，同时可针对测试目的设置不同的实现方式。对此，下列代码显示了此类机制的通用实现方案（将其置于数据中）：

```
package com.sample.marvelgallery.data

abstract class Provider<T> {

   abstract fun creator(): T
```

```
    private val instance: T by lazy { creator() }
    var testingInstance: T? = null

    fun get(): T = testingInstance ?: instance
}
```



此处并不打算定义自己的 Provider，而是使用 Dependency Injection 库，例如 Dagger 或 Kodein。在 Android 开发中，Dagger 十分常见，但当前示例并不打算使用 Dagger，以避免对不熟悉该库的开发人员带来不必要的麻烦。

针对于此，可令 MarvelRepository 伴生对象提供者扩展上述类，如下所示：

```
package com.sample.marvelgallery.data

import com.sample.marvelgallery.model.MarvelCharacter
import io.reactivex.Single

interface MarvelRepository {

    fun getAllCharacters(): Single<List<MarvelCharacter>>

    companion object : Provider<MarvelRepository>() {
        override fun creator() = MarvelRepositoryImpl()
    }
}
```

根据上述定义，可通过 MarvelRepository 伴生对象获得 MarvelRepository 实例，如下所示：

```
val marvelRepository = MarvelRepository.get()
```

这将表示为 MarvelRepositoryImpl 的延迟实例，直至设置了 testingInstance 属性的非空值，如下所示：

```
MarvelRepository.get() // Returns instance of MarvelRepositoryImpl

MarvelRepository.testingInstance= object: MarvelRepository {
    override fun getAllCharacters(): Single<List<MarvelCharacter>>
          = Single.just(emptyList())
}

MarvelRepository.get()//returns an instance of an anonymous class in
which the returned list is always empty.
```

此类构造方式十分有用，并支持基于的 UI 测试。相应地，项目中体现了针对元素覆写的具体应用，读者可在 GitHub 上对其进行查看。本节并未对此予以展示，对于不太熟悉测试机制的开发人员来讲，这也降低了理解难度。对此，读者可访问 https://github.com/MarcinMoskala/MarvelGallery/blob/ master/app/src/androidTest/java/com/sample/。

最后，通过人物角色图片库的显示业务逻辑实现，可将存储库与视图进行连接。

3. 业务逻辑实现

前述内容实现了视图和存储库部分，下面将考察最终的业务逻辑。针对于此，需要获得人物角色列表，并在用户进入相关场景或刷新屏幕时显示列表项。通过 MVP 模式，可从视图实现中获得这一类业务逻辑，其精简后的规则如下所示。

- 模型层：该层负责管理数据。其中，模型的职责包括使用 API、缓存数据以及管理数据库等。
- 显示层：该层表示为模型和视图之间的中间人，且应包含全部显示逻辑。显示层负责响应于用户的交互操作，应用并更新模型和视图。
- 视图层：该层负责显示数据，并将用户交互事件转发至显示层。

在该模式实现中，可将 Activity 视作视图，且针对每个视图，需要生成一个显示层。一种较好的操作方式是编写单元测试，以检测业务逻辑规则是否正确实现。为了简化操作，需要在易于模拟的接口之后隐藏 Activity，以显示与视图（Activity）间的、所有可能的显示层交互。另外，此处还将创建 Activity 中的全部依赖项（例如 MarvelRepository），并作为隐藏在接口后的对象，通过构造函数将其传递至显示层中（例如，作为 Marvel Repository 传递 MarvelRepositoryImpl）。

在显示层中，需要实现下列行为：

- 当显示层等待某个响应时，将显示加载动画。
- 在创建视图后，人物角色列表将被加载和显示。
- 在调用刷新方法后，人物角色列表将被加载。
- 当 API 返回人物角色列表后，将在当前视图上被显示。
- 当 API 返回一个错误时，将在当前视图上显示。

不难发现，显示层须通过构造函数获取视图和 MarvelRepository，同时还应指定相关方法，并在生成视图或用户请求被刷新时被调用，如下所示：

```
package com.sample.marvelgallery.presenter

import com.sample.marvelgallery.data.MarvelRepository
import com.sample.marvelgallery.view.main.MainView
```

```
class MainPresenter(val view: MainView, val repository: MarvelRepository)
{

   fun onViewCreated() {
   }

   fun onRefresh() {
   }
}
```

其中，视图需要定义对应方法，用于显示人物角色列表、显示错误消息，并在视图被刷新时显示进度条（将其定义于 view/main 中，并将 MainActivity 移至 view/main 中），如下所示：

```
package com.sample.marvelgallery.view.main.main

import com.sample.marvelgallery.model.MarvelCharacter

interface MainView {
   var refresh: Boolean
   fun show(items: List<MarvelCharacter>)
   fun showError(error: Throwable)
}
```

在向显示层添加逻辑之前，下面首先定义两个单元测试，如下所示：

```
// test source set
package com.sample.marvelgallery

import com.sample.marvelgallery.data.MarvelRepository
import com.sample.marvelgallery.model.MarvelCharacter
import com.sample.marvelgallery.presenter.MainPresenter
import com.sample.marvelgallery.view.main.MainView
import io.reactivex.Single
import org.junit.Assert.assertEquals
import org.junit.Assert.fail
import org.junit.Test

@Suppress("IllegalIdentifier") // 1
class MainPresenterTest {

   @Test
   fun `After view was created, list of characters is loaded and
displayed`() {
```

```
    assertOnAction { onViewCreated() }.thereIsSameListDisplayed()
}

@Test
fun `New list is shown after view was refreshed`() {
    assertOnAction { onRefresh() }.thereIsSameListDisplayed()
}

private fun assertOnAction(action: MainPresenter.() -> Unit)
        = PresenterActionAssertion(action)

private class PresenterActionAssertion
(val actionOnPresenter: MainPresenter.() -> Unit) {

fun thereIsSameListDisplayed() {
    // Given
    val exampleCharacterList = listOf(// 2
            MarvelCharacter("ExampleName", "ExampleImageUrl"),
            MarvelCharacter("Name1", "ImageUrl1"),
            MarvelCharacter("Name2", "ImageUrl2")
    )

    var displayedList: List<MarvelCharacter>? = null

    val view = object : MainView { //3
        override var refresh: Boolean = false

        override fun show(items: List<MarvelCharacter>) {
            displayedList = items // 4
        }

        override fun showError(error: Throwable) {
            fail() //5
        }
    }

    val marvelRepository = object : MarvelRepository { // 3
        override fun getAllCharacters():
         Single<List<MarvelCharacter>>
           = Single.just(exampleCharacterList) // 6
    }

    val mainPresenter = MainPresenter(view, marvelRepository)
```

```
        // 3

        // When
        mainPresenter.actionOnPresenter() // 7

        // Then
        assertEquals(exampleCharacterList, displayedList) // 8

     }
   }
}
```

对于注释 1，Kotlin 单元测试中允许使用描述名称，但这将显示一条警告消息。对此，须隐藏该警告消息；对于注释 2，定义一个可显示的人物角色列表；对于注释 3，定义视图和存储库，并以此创建一个显示层；对于注释 4，当显示元素列表时，应将其设置为可显示列表；对于注释 5，若 showError 被调用，测试将失败；对于注释 6，getAll Characters 方法返回样例列表；对于注释 7，在显示层上调用一个定义完毕的行为；对于注释 8，检测存储库返回的列表是否与所显示的列表相同。

为了简化上述定义，可析取出 BaseMarvelRepository 和 BaseMainView，并在独立类中持有样例数据，如下所示：

```
// test source set
package com.sample.marvelgallery.helpers

import com.sample.marvelgallery.data.MarvelRepository
import com.sample.marvelgallery.model.MarvelCharacter
import io.reactivex.Single

class BaseMarvelRepository(
       val onGetCharacters: () -> Single<List<MarvelCharacter>>
) : MarvelRepository {

   override fun getAllCharacters() = onGetCharacters()
}

// test source set
package com.sample.marvelgallery.helpers

import com.sample.marvelgallery.model.MarvelCharacter
import com.sample.marvelgallery.view.main.MainView

class BaseMainView(
```

```
        var onShow: (items: List<MarvelCharacter>) -> Unit = {},
        val onShowError: (error: Throwable) -> Unit = {},
        override var refresh: Boolean = false
) : MainView {

    override fun show(items: List<MarvelCharacter>) {
        onShow(items)
    }

    override fun showError(error: Throwable) {
        onShowError(error)
    }
}

// test source set
package com.sample.marvelgallery.helpers

import com.sample.marvelgallery.model.MarvelCharacter

object Example {
    val exampleCharacter = MarvelCharacter
    ("ExampleName", "ExampleImageUrl")
    val exampleCharacterList = listOf(
            exampleCharacter,
            MarvelCharacter("Name1", "ImageUrl1"),
            MarvelCharacter("Name2", "ImageUrl2")
    )
}
```

当前，可简化 PresenterActionAssertion 的定义，如下所示：

```
package com.sample.marvelgallery

import com.sample.marvelgallery.helpers.BaseMainView
import com.sample.marvelgallery.helpers.BaseMarvelRepository
import com.sample.marvelgallery.helpers.Example
import com.sample.marvelgallery.model.MarvelCharacter
import com.sample.marvelgallery.presenter.MainPresenter
import io.reactivex.Single
import org.junit.Assert.assertEquals
import org.junit.Assert.fail
import org.junit.Test

@Suppress("IllegalIdentifier")
```

```
class MainPresenterTest {

   @Test
   fun `After view was created, list of characters is loaded and
       displayed`() {
       assertOnAction { onViewCreated() }.thereIsSameListDisplayed()
   }

   @Test
   fun `New list is shown after view was refreshed`() {
       assertOnAction { onRefresh() }.thereIsSameListDisplayed()
   }

   private fun assertOnAction(action: MainPresenter.() -> Unit)
           = PresenterActionAssertion(action)

   private class PresenterActionAssertion
   (val actionOnPresenter: MainPresenter.() -> Unit) {

       fun thereIsSameListDisplayed() {
           // Given
           var displayedList: List<MarvelCharacter>? = null

           val view = BaseMainView(
                   onShow = { items -> displayedList = items },
                   onShowError = { fail() }
           )

           val marvelRepository = BaseMarvelRepository(
                 onGetCharacters =
           { Single.just(Example.exampleCharacterList) }
           )

           val mainPresenter = MainPresenter(view, marvelRepository)

           // When
           mainPresenter.actionOnPresenter()

           // Then
           assertEquals(Example.exampleCharacterList, displayedList)
       }
   }
}
```

当启动测试时，对应效果如图 9.13 所示。

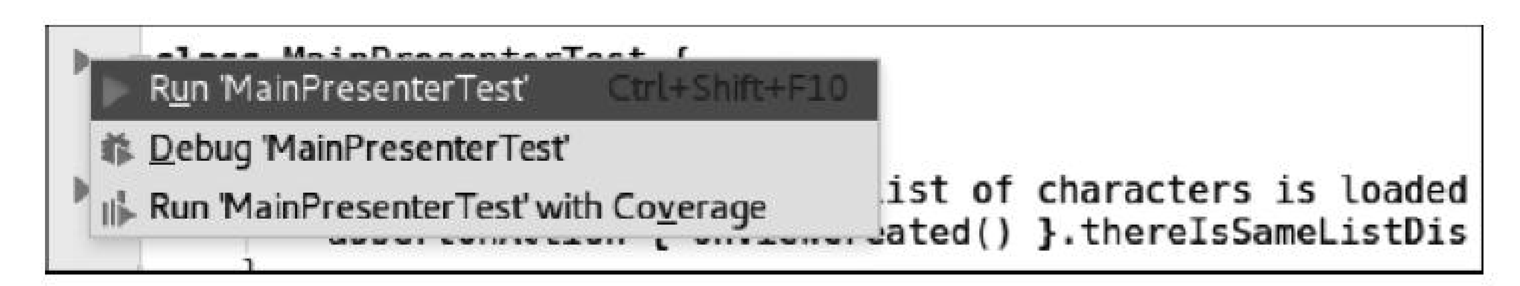


图 9.13

从图 9.14 中可知，测试并未通过。

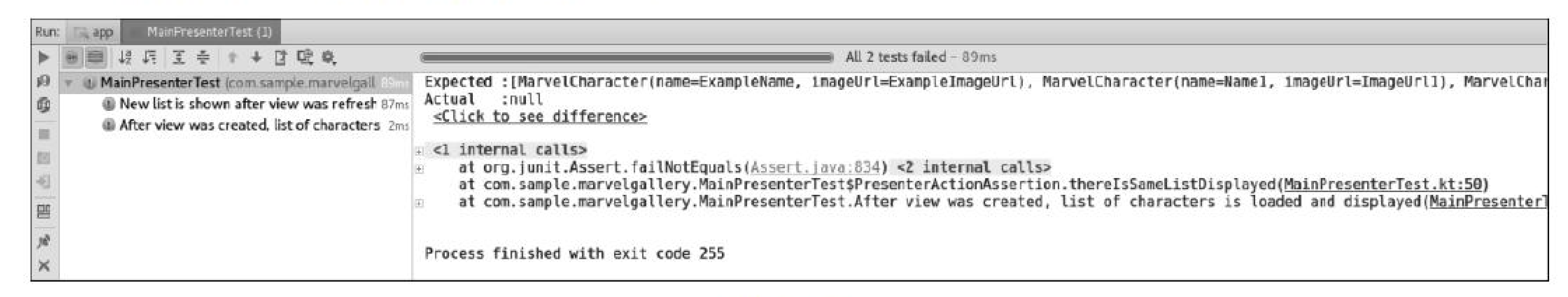


图 9.14

其原因在于，相关功能项并未在 MainPresenter 中予以实现。对此，满足单元测试的最简单的代码如下所示：

```
package com.sample.marvelgallery.presenter

import com.sample.marvelgallery.data.MarvelRepository
import com.sample.marvelgallery.view.main.MainView

class MainPresenter(val view: MainView, val repository: MarvelRepository)
{

   fun onViewCreated() {
       loadCharacters()
}

   fun onRefresh() {
       loadCharacters()
   }

   private fun loadCharacters() {
       repository.getAllCharacters()
              .subscribe({ items ->
                  view.show(items)
              })
   }
}
```

图 9.15 显示了通过测试后的效果。



图 9.15

但针对下列实现，仍存在两个问题：

● 由于 getAllCharacters 使用了网络操作，且无法像当前示例中那样运行于主线程上，因而无法在 Android 中正常工作。
● 如果用户在加载完成之前离开当前应用程序，将存在内存泄漏问题。
● 对于第一个问题，需要指定操作的具体线程。网络线程应运行于 I/O 线程上；同时还应在 Android 的主线程上进行查看（由于在回调中改变了视图），如下所示：

```
repository.getAllCharacters()
        .subscribeOn(Schedulers.io()) // 1
        .observeOn(AndroidSchedulers.mainThread()) // 2
        .subscribe({ items -> view.show(items) })
```

针对注释 1，网络线程应运行于 I/O 线程上；对于注释 2，回调应在主线程上启用。

对于这一类较为常见的调度程序，可将其设置于顶级扩展函数中，如下所示：

```
// RxExt.kt
package com.sample.marvelgallery.data

import io.reactivex.Single
import io.reactivex.android.schedulers.AndroidSchedulers
import io.reactivex.schedulers.Schedulers

fun <T> Single<T>.applySchedulers(): Single<T> = this
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())

And use it in MainPresenter:

repository.getAllCharacters()
        .applySchedulers()
        .subscribe({ items -> view.show(items) })
```

测试不允许访问 Android 主线程。因此，测试无法通过。另外，运行于新线程上的操作并非是单元测试中的所需内容，这会产生断言同步问题。当解决此类问题时，须在单元

测试之前覆写调度程序，并使全部内容运行于同一线程上（将其添加至 Main PresenterTest 类中），如下所示：

```
package com.sample.marvelgallery

import com.sample.marvelgallery.helpers.BaseMainView
import com.sample.marvelgallery.helpers.BaseMarvelRepository
import com.sample.marvelgallery.helpers.Example
import com.sample.marvelgallery.model.MarvelCharacter
import com.sample.marvelgallery.presenter.MainPresenter
import io.reactivex.Single
import io.reactivex.android.plugins.RxAndroidPlugins
import io.reactivex.plugins.RxJavaPlugins
import io.reactivex.schedulers.Schedulers
import org.junit.Assert.assertEquals
import org.junit.Assert.fail
import org.junit.Before
import org.junit.Test

@Suppress("IllegalIdentifier")

class MainPresenterTest {

   @Before
fun setUp() {
    RxAndroidPlugins.setInitMainThreadSchedulerHandler {
        Schedulers.trampoline() }
    RxJavaPlugins.setIoSchedulerHandler { Schedulers.trampoline() }
    RxJavaPlugins.setComputationSchedulerHandler {
        Schedulers.trampoline() }
    RxJavaPlugins.setNewThreadSchedulerHandler {
        Schedulers.trampoline() }
}

@Test
fun `After view was created, list of characters is loaded and
     displayed`() {
    assertOnAction { onViewCreated() }.thereIsSameListDisplayed()
}

@Test
fun `New list is shown after view was refreshed`() {
    assertOnAction { onRefresh() }.thereIsSameListDisplayed()
}
```

如图 9.16 所示，测试再次通过。

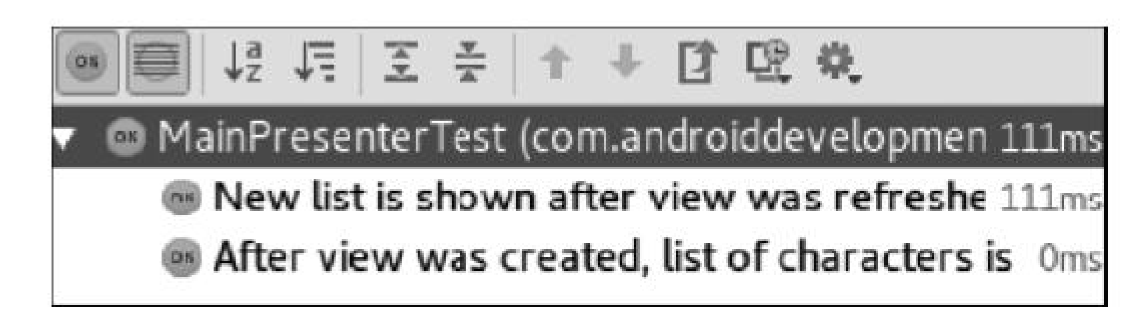


图 9.16

另一个问题则是内存泄漏，也就是说，用户在获取服务器响应之前离开当前应用程序。一类常见的方法是，当用户离开应用程序时，使全部订阅处于复合状态，并对其进行处理，如下所示：

```
private var subscriptions = CompositeDisposable()

fun onViewDestroyed() {
   subscriptions.dispose()
}
```

在较大的应用程序项目中，大多数显示层均包含一些订阅服务。因此，收集订阅服务，以及当用户销毁视图时的处理行为均可视作一种常见行为，并可从 BasePresenter 中获取。除此之外，为了简化操作过程，还可定义 BaseActivityWithPresenter 类，以加载 Presenter 接口背后的显示层，进而在视图被销毁时调用 onViewDestroyed 方法。下面在应用程序中实现这一机制，Presenter 的定义如下所示：

```
package com.sample.marvelgallery.presenter

interface Presenter {
   fun onViewDestroyed()
}
```

BasePresenter 的定义如下所示：

```
package com.sample.marvelgallery.presenter

import io.reactivex.disposables.CompositeDisposable

abstract class BasePresenter : Presenter {

   protected var subscriptions = CompositeDisposable()

   override fun onViewDestroyed() {
       subscriptions.dispose()
   }
}
```

BaseActivityWithPresenter 的定义如下所示：

```
package com.sample.marvelgallery.view.common

import android.support.v7.app.AppCompatActivity
import com.sample.marvelgallery.presenter.Presenter

abstract class BaseActivityWithPresenter : AppCompatActivity() {

   abstract val presenter: Presenter

   override fun onDestroy() {
       super.onDestroy()
       presenter.onViewDestroyed()
   }
}
```

为了简化订阅服务的添加操作，可定义一个 plusAssgin 操作符，如下所示：

```
// RxExt.ext
package com.sample.marvelgallery.data

import io.reactivex.Single
import io.reactivex.android.schedulers.AndroidSchedulers
import io.reactivex.disposables.CompositeDisposable
import io.reactivex.disposables.Disposable
import io.reactivex.schedulers.Schedulers

fun <T> Single<T>.applySchedulers(): Single<T> = this
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())

operator fun CompositeDisposable.plusAssign(disposable: Disposable) {
   add(disposable)
}
```

根据上述内容，可确保 MainPresenter 处于安全状态，如下所示：

```
package com.sample.marvelgallery.presenter

import com.sample.marvelgallery.data.MarvelRepository
import com.sample.marvelgallery.data.applySchedulers
import com.sample.marvelgallery.data.plusAssign
import com.sample.marvelgallery.view.main.MainView
```

```
class MainPresenter(
        val view: MainView,
        val repository: MarvelRepository
) : BasePresenter() {

   fun onViewCreated() {
       loadCharacters()
   }

   fun onRefresh() {
       loadCharacters()
   }

   private fun loadCharacters() {
       subscriptions += repository.getAllCharacters()
               .applySchedulers()
               .subscribe({ items ->
                   view.show(items)
               })

   }
}
```

其中，前两个 MainPresenter 行为均已实现，当 API 返回一条错误消息时，对应内容将显示于视图上，此处可将这一条件作为一项测试添加至 MainPresenterTest 中，如下所示：

```
@Test
fun `New list is shown after view was refreshed`() {
   assertOnAction { onRefresh() }.thereIsSameListDisplayed()
}

@Test
fun `When API returns error, it is displayed on view`() {
   // Given
   val someError = Error()
   var errorDisplayed: Throwable? = null
   val view = BaseMainView(
           onShow = {   -> fail() },
           onShowError = { errorDisplayed = it }
   )
   val marvelRepository = BaseMarvelRepository
   { Single.error(someError) }
   val mainPresenter = MainPresenter(view, marvelRepository)
```

```
    // When
    mainPresenter.onViewCreated()
    // Then
    assertEquals(someError, errorDisplayed)
}

private fun assertOnAction(action: MainPresenter.() -> Unit)
        = PresenterActionAssertion(action)
```

稍作变化即可使该测试通过，即 MainPresenter 的 subscribe 方法中的错误处理程序规范，如下所示：

```
subscriptions += repository.getAllCharacters()
        .applySchedulers()
        .subscribe({ items -> // onNext
            view.show(items)
        }, { // onError
            view.showError(it)
        })
```

虽然 subscribe 是 Java 中的方法，但我们无法使用命名参数规则。此类调用并非真正具备描述性特征，这也是在 RxExt.kt 中设置名为 subscribeBy 的自定义方法的原因，如下所示：

```
// Ext.kt

fun <T> Single<T>.applySchedulers(): Single<T> = this
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
fun <T> Single<T>.subscribeBy(
        onError: ((Throwable) -> Unit)? = null,
        onSuccess: (T) -> Unit
): Disposable = subscribe(onSuccess, { onError?.invoke(it) })
```

因而此处将不再使用订阅服务，实际操作如下所示：

```
subscriptions += repository.getAllCharacters()
        .applySchedulers()
        .subscribeBy(
                onSuccess = view::show,
                onError = view::showError
      )
```



针对不同 RxJava 类型所定义的完整版本的 subscribeBy（Observable、Flowable 等），以及面向 RxJava 的其他有效 Kotlin 扩展位于 RxKotlin 库中（对应网址为 https://github.com/ReactiveX/RxKotlin）。

当显示或隐藏加载过程时，可对出现于处理前/后的事件定义额外的监听器，如下所示：

```
subscriptions += repository.getAllCharacters()
        .applySchedulers()
        .doOnSubscribe { view.refresh = true },}
                onSuccess = view::show,
        .doFinally { view.refresh = false }
        .subscribeBy(
                onSuccess = view::show,
                onError = view::showError,
                onFinish = { view.refresh = false }
        )
```

随后，测试再次通过，如图 9.17 所示。

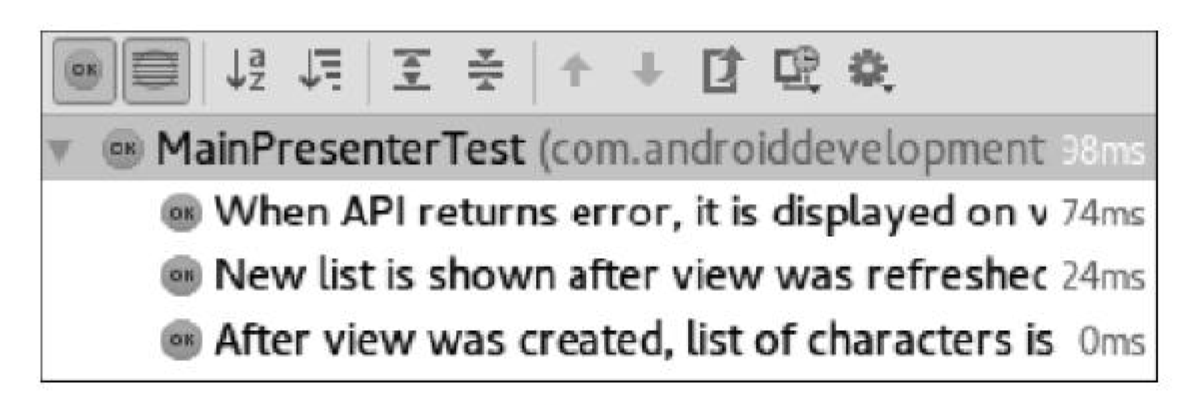


图 9.17

当前，subscribe 方法的可读性变得越来越差，但可协同另一个商业逻辑处理此类问题，其定义可描述为：当显示层迭代某个响应时，刷新将被显示。下列代码在 Main PresenterTest 中定义了单元测试。

```
package com.sample.marvelgallery

import com.sample.marvelgallery.helpers.BaseMainView
import com.sample.marvelgallery.helpers.BaseMarvelRepository
import com.sample.marvelgallery.helpers.Example
import com.sample.marvelgallery.model.MarvelCharacter
import com.sample.marvelgallery.presenter.MainPresenter
import io.reactivex.Single
import io.reactivex.android.plugins.RxAndroidPlugins
import io.reactivex.plugins.RxJavaPlugins
import io.reactivex.schedulers.Schedulers
```

```
import org.junit.Assert.*
import org.junit.Before
import org.junit.Test

@Suppress("IllegalIdentifier")

class MainPresenterTest {

   @Test
   fun `When presenter is waiting for response, refresh is displayed`()
   {
      // Given
      val view = BaseMainView(refresh = false)
      val marvelRepository = BaseMarvelRepository(
              onGetCharacters = {
                  Single.fromCallable {
                      // Then
                      assertTrue(view.refresh) // 1
                      Example.exampleCharacterList
                  }
              }
      )
      val mainPresenter = MainPresenter(view, marvelRepository)
      view.onShow = {   ->
          // Then
          assertTrue(view.refresh) // 1
      }
      // When
      mainPresenter.onViewCreated()
      // Then
      assertFalse(view.refresh) // 1
  }
}
```

对于注释 1，希望在网络请求过程中以及显示元素时显示刷新，而非处理结束之后。



在 RxJava2 上的版本中，回调中的断言并未中断测试，但却在执行报告中显示了一条错误消息，如图 9.18 和图 9.19 所示。



图 9.18



图 9.19



在未来版本中，可能会加入一个处理程序，并可从回调内部中断某个测试。

当显示或隐藏加载过程时，可对出现于处理前/后的事件定义额外的监听器，如下所示：

```
subscriptions += repository.getAllCharacters()
        .applySchedulers()
        .doOnSubscribe { view.refresh = true }
        .doFinally { view.refresh = false }
        .subscribeBy(
                        onSuccess = view::show,
                        onError = view::showError
        )
```

在经历了上述全部调整后，全部测试均可再次通过，如图 9.20 所示。



图 9.20

至此，我们得到了完整的功能显示层、网络以及视图，可将其进行适当整合并完成第一个用例。

4. 整合

目前，项目中的 MainPresenter 已准备完毕，并可在 MainActivity 中对其加以使用，如下所示：

```
package com.sample.marvelgallery.view.main

import android.os.Bundle
import android.support.v7.widget.GridLayoutManager
import android.view.Window
import com.sample.marvelgallery.R
import com.sample.marvelgallery.data.MarvelRepository
import com.sample.marvelgallery.model.MarvelCharacter
import com.sample.marvelgallery.presenter.MainPresenter
import com.sample.marvelgallery.view.common.BaseActivityWithPresenter
import com.sample.marvelgallery.view.common.bindToSwipeRefresh
import com.sample.marvelgallery.view.common.toast
import kotlinx.android.synthetic.main.activity main.*

class MainActivity : BaseActivityWithPresenter(), MainView { // 1

   override var refresh by bindToSwipeRefresh(R.id.swipeRefreshView)
   // 2
   override val presenter by lazy
   { MainPresenter(this, MarvelRepository.get()) } // 3

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       requestWindowFeature(Window.FEATURE NO TITLE)
       setContentView(R.layout.activity main)
       recyclerView.layoutManager = GridLayoutManager(this, 2)
       swipeRefreshView.setOnRefreshListener
       { presenter.onRefresh() } // 4
       presenter.onViewCreated() // 4
  }

  override fun show(items: List<MarvelCharacter>) {
      val categoryItemAdapters = items.map(::CharacterItemAdapter)
      recyclerView.adapter = MainListAdapter(categoryItemAdapters)
  }

  override fun showError(error: Throwable) {
      toast("Error: ${error.message}") // 2
      error.printStackTrace()
  }
}
```

对于注释 1，Activity 应扩展 BaseActivityWithPresenter 并实现 MainView；对于注释 2，bindToSwipeRefresh 和 toast 尚未实现；对于注释 3，此处通过延迟方式定义显示层。其中，第一个参数表示为指向 MainView 接口之后的 Activity 的引用；对于注释 4，须通过其方法向显示层传递事件。

在上述代码中，使用了之前已经讨论的两个函数，toast（用于显示屏幕上的烤面包）和 bindToSwipeRefresh（用于将属性与划动手势的可见性进行绑定），对应内容如下所示：

```
// ViewExt.kt
package com.sample.marvelgallery.view.common

import android.app.Activity
import android.content.Context
import android.support.annotation.IdRes
import android.support.v4.widget.SwipeRefreshLayout
import android.support.v7.widget.RecyclerView
import android.view.View
import android.widget.ImageView
import android.widget.Toast
import com.bumptech.glide.Glide
import kotlin.properties.ReadWriteProperty
import kotlin.reflect.KProperty

fun <T : View> RecyclerView.ViewHolder.bindView(viewId: Int)
       = lazy { itemView.findViewById<T>(viewId) }

fun ImageView.loadImage(photoUrl: String) {
   Glide.with(context)
           .load(photoUrl)
           .into(this)
}

fun Context.toast(text: String, length: Int = Toast.LENGTH LONG) {
   Toast.makeText(this, text, length).show()
}

fun Activity.bindToSwipeRefresh(@IdRes swipeRefreshLayoutId: Int):
ReadWriteProperty<Any?, Boolean>
       = SwipeRefreshBinding(lazy {
findViewById<SwipeRefreshLayout>(swipeRefreshLayoutId) })

private class SwipeRefreshBinding(lazyViewProvider:
Lazy<SwipeRefreshLayout>) : ReadWriteProperty<Any?, Boolean> {
```

```
    val view by lazyViewProvider

    override fun getValue(thisRef: Any?,
    property: KProperty<*>): Boolean {
        return view.isRefreshing
    }

    override fun setValue(thisRef: Any?,
    property: KProperty<*>, value: Boolean) {
        view.isRefreshing = value
    }
}
```

图 9.21 显示了正确的人物角色列表。

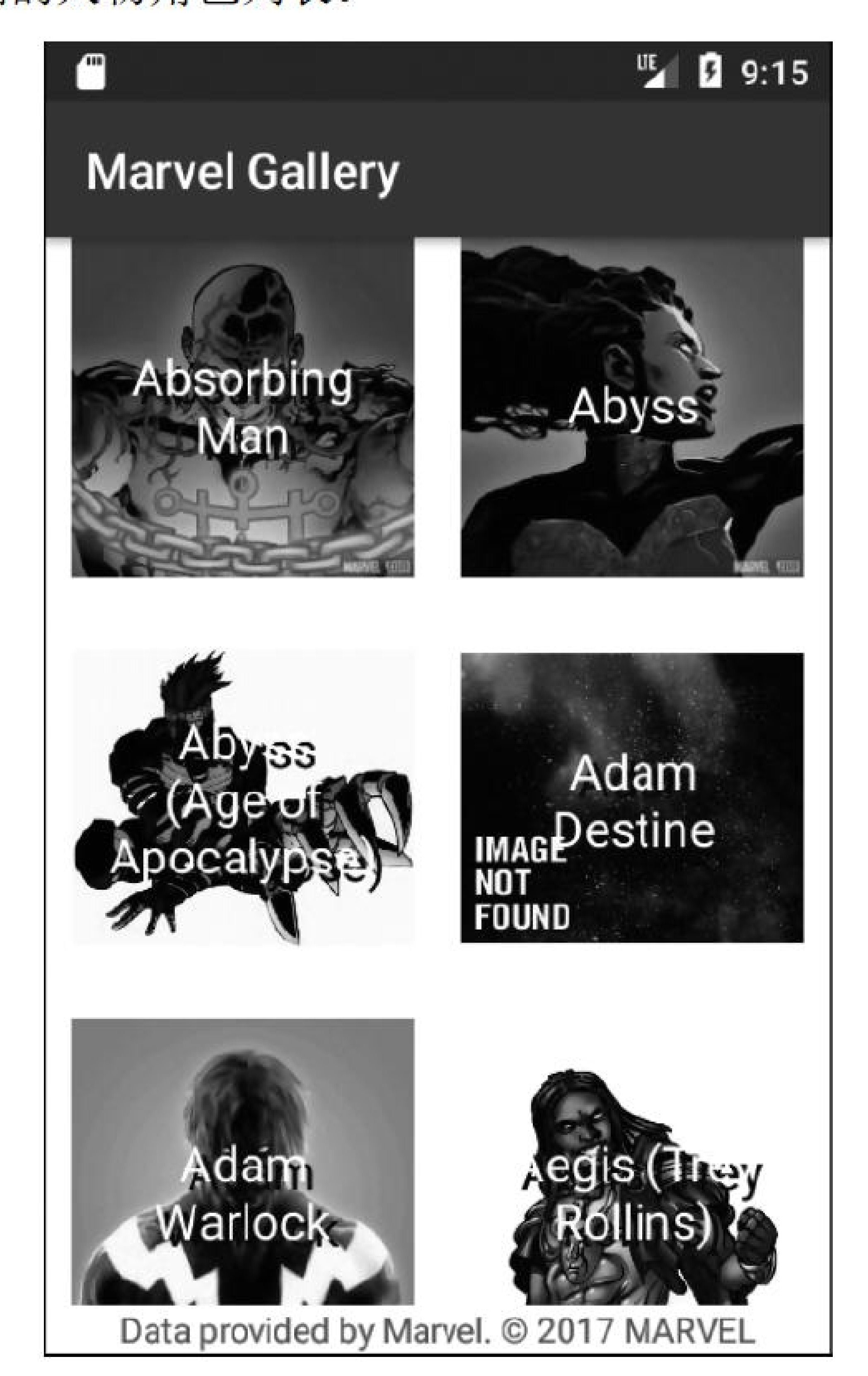


图 9.21

至此，第一个用例讨论完毕，下面讨论人物角色搜索问题。

9.1.4　人物角色搜索

另一个需要实现的操作是人物角色的搜索功能，该用例可描述为：在启动应用程序后，用户可根据名称搜索某个人物角色。

当添加此项功能时，可向 activity_main 布局中添加 EditText，如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:id="@+id/charactersView"
   android:layout width="match parent"
   android:layout height="match parent"
   android:background="@android:color/white"
   android:fitsSystemWindows="true">

<!-- Dummy item to prevent EditText from receiving
     focus on initial load -->
   <LinearLayout
       android:layout width="0px"
       android:layout height="0px"
       android:focusable="true"
       android:focusableInTouchMode="true"
       tools:ignore="UselessLeaf" />

  <android.support.design.widget.TextInputLayout
     android:id="@+id/searchViewLayout"
     android:layout width="match parent"
     android:layout height="wrap content"
     android:layout margin="@dimen/element padding">

     <EditText
         android:id="@+id/searchView"
         android:layout width="match parent"
         android:layout height="wrap content"
         android:layout centerHorizontal="true"
         android:hint="@string/search hint" />
  </android.support.design.widget.TextInputLayout>

   <android.support.v4.widget.SwipeRefreshLayout
xmlns:android="http://schemas.android.com/apk/res/android"
```

```
        android:id="@+id/swipeRefreshView"
        android:layout width="match parent"
        android:layout height="match parent"
        android:layout below="@+id/searchViewLayout"
        app:layout behavior="@string/appbar scrolling view behavior">

        <android.support.v7.widget.RecyclerView
            android:id="@+id/recyclerView"
            android:layout width="match parent"
            android:layout height="match parent"
            android:scrollbars="vertical" />
    </android.support.v4.widget.SwipeRefreshLayout>

    <TextView
        android:layout width="match parent"
        android:layout height="wrap content"
        android:layout alignParentBottom="true"
        android:background="@android:color/white"
        android:gravity="center"
        android:text="@string/marvel copyright notice" />
</RelativeLayout>
```

另外，还须添加 Android 兼容库依赖项，进而可使用 TextInputLayout，如下所示：

```
implementation
"com.android.support:appcompat-v7:$android support version"
implementation "com.android.support:design:$android support version"
implementation "com.android.support:recyclerview
v7:$android_support_version"
```

strings.xml 文件中的 search_hint 字符串定义如下所示：

```
<resources>
    <string name="app name">MarvelGallery</string>
    <string name="search hint">Search for character</string>
    <string name="marvel copyright notice">
        Data provided by Marvel. . 2017 MARVEL
    </string>
</resources>
```

除此之外，还须设置相关标签，并在开启键盘时显示 Marvel 版权信息；同时，还须在 AndroidManifest 中自动调整 activity 定义中的 windowSoftInputMode，如下所示：

```
<activity
    android:name="com.sample.marvelgallery.view.main.MainActivity"
```

```
    android:windowSoftInputMode="adjustResize">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

对应结果如图 9.22 所示。

下面可将搜索栏添加至 MainActivity 中，如图 9.23 所示。



图 9.22

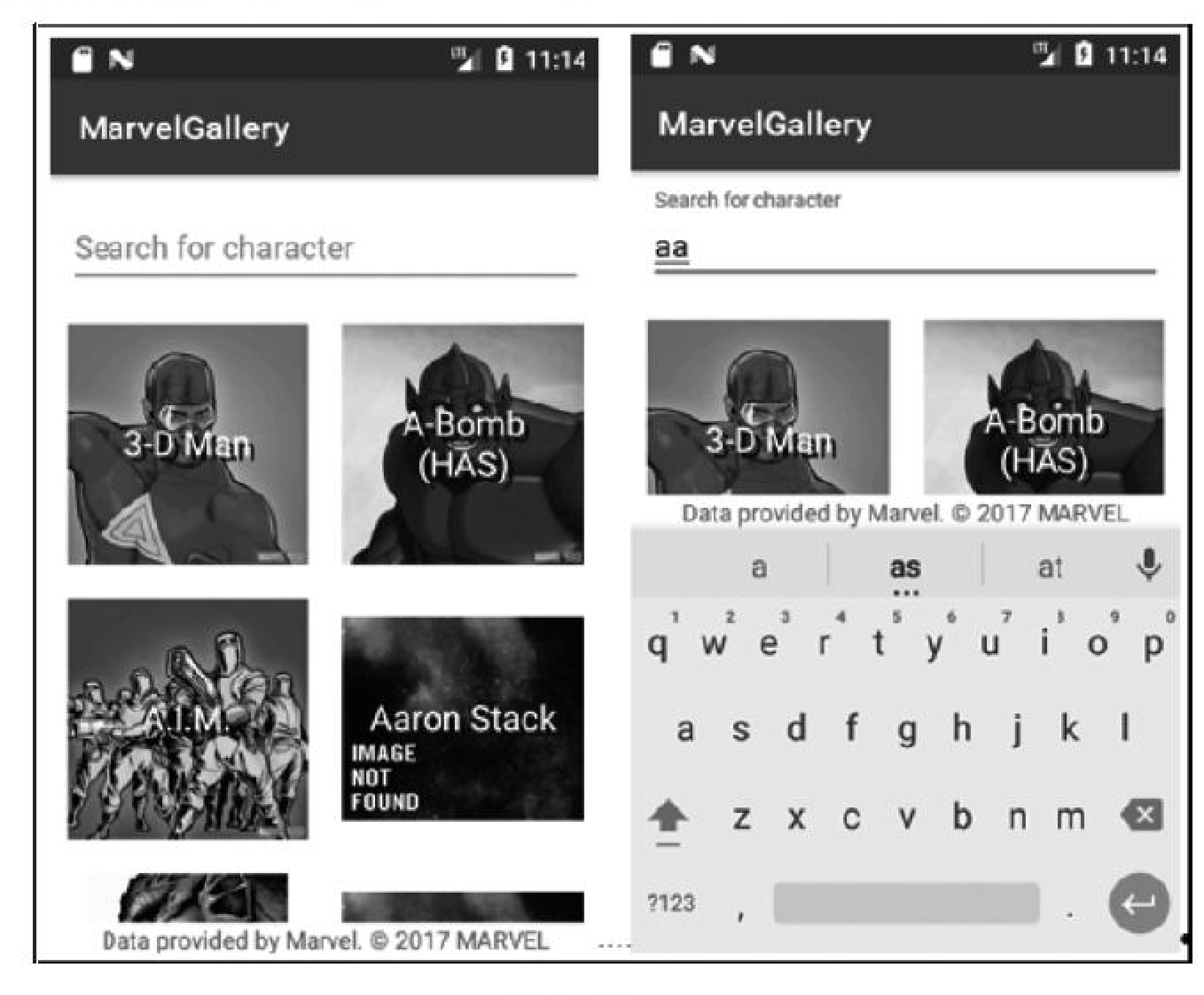


图 9.23

此处所期望的行为可描述为：当用户修改搜索栏中的文本时，将加载一个新列表。对此，须在 MainPresenter 中定义一个新方法，用于通知显示层文本方式变化。该方法称作 onSearchChanged，相关内容如下所示：

```
fun onRefresh() {
    loadCharacters()
}

fun onSearchChanged(text: String) {
    // TODO
}

private fun loadCharacters() {
    subscriptions += repository.getAllCharacters()
            .applySchedulers()
```

```
            .doOnSubscribe { view.refresh = true }
            .doFinally { view.refresh = false }
            .subscribeBy(
                onSuccess = view::show,
                onError = view::showError
          )
    }
}
```

此处需要修改 MarvelRepository 定义，并作为 getAllCharacters 参数接收一个搜索查询（需要注意的是，还须更新 BaseMarvelRepository），如下所示：

```
interface MarvelRepository {

   fun getAllCharacters(searchQuery: String?):
   Single<List<MarvelCharacter>>

   companion object : Provider<MarvelRepository>() {
       override fun creator() = MarvelRepositoryImpl()
   }
}
```

最后，更新后的实现过程如下所示：

```
class MarvelRepositoryImpl : MarvelRepository {

   val api = retrofit.create(MarvelApi::class.java)

   override fun getAllCharacters(searchQuery: String?):
   Single<List<MarvelCharacter>> = api.getCharacters(
           offset = 0,
           searchQuery = searchQuery,
           limit = elementsOnListLimit
   ).map { it.data?.results.orEmpty().map(::MarvelCharacter) ?:
    emptyList() }

   companion object {
       const val elementsOnListLimit = 50
   }
}
```

另外，还须更新网络请求定义，如下所示：

```
interface MarvelApi {
```

```
    @GET("characters")
    fun getCharacters(
            @Query("offset") offset: Int?,
            @Query("nameStartsWith") searchQuery: String?,
            @Query("limit") limit: Int?
    ): Single<DataWrapper<List<CharacterMarvelDto>>>
}
```

为了对代码进行编译，应提供 null 作为 MainPresenter 中的 getAllCharacters 参数，如下所示：

```
private fun loadCharacters() {
    subscriptions += repository.getAllCharacters(null)
            .applySchedulers()
            .doOnSubscribe { view.refresh = true }
            .doFinally { view.refresh = false }
            .subscribeBy(
                    onSuccess = view::show,
                    onError = view::showError
          )
    }
}
```

同时，还须更新 BaseMarvelRepository，如下所示：

```
package com.sample.marvelgallery.helpers

import com.sample.marvelgallery.data.MarvelRepository
import com.sample.marvelgallery.model.MarvelCharacter
import io.reactivex.Single

class BaseMarvelRepository(
       val onGetCharacters: (String?) -> Single<List<MarvelCharacter>>
) : MarvelRepository {

    override fun getAllCharacters(searchQuery: String?)
            = onGetCharacters(searchQuery)
}
```

当前，网络实现将返回一个源自查询的人物角色列表；或者，若未指定查询，则返回一个填充列表。对此，可首先定义显示层，下面定义相关测试：

```
  @file:Suppress("IllegalIdentifier")

  package com.sample.marvelgallery
```

```
import com.sample.marvelgallery.helpers.BaseMainView
import com.sample.marvelgallery.helpers.BaseMarvelRepository
import com.sample.marvelgallery.presenter.MainPresenter
import io.reactivex.Single
import org.junit.Assert.*
import org.junit.Test

class MainPresenterSearchTest {
   @Test
   fun `When view is created, then search query is null`() {
       assertOnAction { onViewCreated() } searchQueryIsEqualTo null
   }

   @Test
   fun `When text is changed, then we are searching for new query`() {
       for (text in listOf("KKO", "HJ HJ", "And so what?"))
           assertOnAction { onSearchChanged(text) }
           searchQueryIsEqualTo text
   }

   private fun assertOnAction(action: MainPresenter.() -> Unit)
         = PresenterActionAssertion(action)

   private class PresenterActionAssertion(val actionOnPresenter:
       MainPresenter.() -> Unit) {

       infix fun searchQueryIsEqualTo(expectedQuery: String?) {
           var checkApplied = false
           val view = BaseMainView(onShowError = { fail() })
           val marvelRepository = BaseMarvelRepository { searchQuery ->
               assertEquals(expectedQuery, searchQuery)
               checkApplied = true
               Single.never()
           }
           val mainPresenter = MainPresenter(view, marvelRepository)
           mainPresenter.actionOnPresenter()
           assertTrue(checkApplied)
       }
   }
}
```

为了使测试通过，需要添加搜索查询，并作为参数（默认参数）向 MainPresenter 的

loadCharacters 方法添加搜索查询，如下所示：

```
fun onSearchChanged(text: String) {
   loadCharacters(text)
}

private fun loadCharacters(searchQuery: String? = null) {
   subscriptions += repository.getAllCharacters(searchQuery)
            .applySchedulers()
            .doOnSubscribe { view.refresh = true }
            .doFinally { view.refresh = false }
            .subscribeBy(
                        onSuccess = view::show,
                        onError = view::showError
         )
   }
}
```

这里的问题在于，Marvel API 不允许空格作为搜索查询内容，此时将会发送 null。因此，如果用户删除了最后一个字符，或者试图在搜索栏中查找空格，程序将会崩溃，应防止此类情况出现。下列测试将检查显示层是否将仅包含空格键的查询转换为 null。

```
@Test
fun `When text is changed, then we are searching for new query`() {
   for (text in listOf("KKO", "HJ HJ", "And so what?"))
       assertOnAction { onSearchChanged(text) }
       searchQueryIsEqualTo text
}

@Test
fun `For blank text, there is request with null query`() {
   for (emptyText in listOf("", " ", "  "))
       assertOnAction { onSearchChanged(emptyText) }
       searchQueryIsEqualTo null
}

private fun assertOnAction(action: MainPresenter.() -> Unit)
      = PresenterActionAssertion(action)
```

在 loadCharacters 方法中，可实现一种安全监测机制，如下所示：

```
private fun loadCharacters(searchQuery: String? = null) {
   val qualifiedSearchQuery = if (searchQuery.isNullOrBlank()) null
```

```
                                  else searchQuery
   subscriptions += repository
           .getAllCharacters(qualifiedSearchQuery)
           .applySchedulers()
           .smartSubscribe(
                   onStart = { view.refresh = true },
                   onSuccess = view::show,
                   onError = view::showError,
                   onFinish = { view.refresh = false }
           )
}
```

当前，全部测试均已通过，如图 9.24 所示。

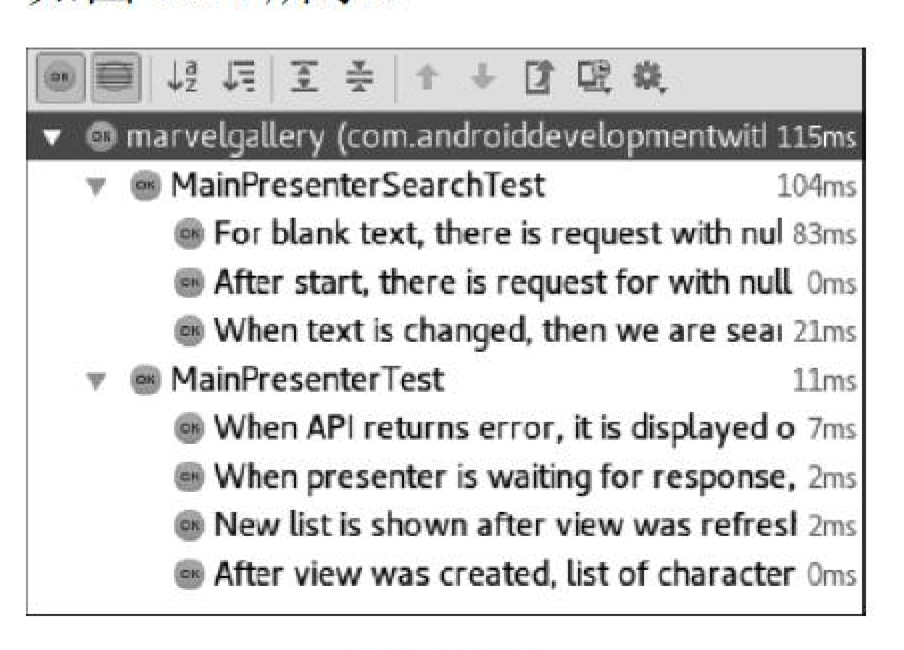


图 9.24

除此之外，还须实现 Activity 功能，并在文本变化时调用显示层。对此，可使用第 7 章讨论的回调类（可选），如下所示：

```
// TextChangedListener.kt
package com.sample.marvelgallery.view.common

import android.text.Editable
import android.text.TextWatcher
import android.widget.TextView

fun TextView.addOnTextChangedListener(config: TextWatcherConfiguration.()
-> Unit) {
   addTextChangedListener(TextWatcherConfiguration().apply { config() }
   addTextChangedListener(textWatcher)
}

class TextWatcherConfiguration : TextWatcher {

   private var beforeTextChangedCallback:
```

```
    (BeforeTextChangedFunction)? = null
    private var onTextChangedCallback:
    (OnTextChangedFunction)? = null
    private var afterTextChangedCallback:
    (AfterTextChangedFunction)? = null

    fun beforeTextChanged(callback: BeforeTextChangedFunction) {
        beforeTextChangedCallback = callback
    }

    fun onTextChanged(callback: OnTextChangedFunction) {
        onTextChangedCallback = callback
    }

    fun afterTextChanged(callback: AfterTextChangedFunction) {
        afterTextChangedCallback = callback
    }

    override fun beforeTextChanged(s: CharSequence,
    start: Int, count: Int, after: Int) {
        beforeTextChangedCallback?.invoke(s.toString(),
        start, count, after)
    }

    override fun onTextChanged(s: CharSequence, start: Int,
    before: Int, count: Int) {
        onTextChangedCallback?.invoke(s.toString(),
        start, before, count)
    }

    override fun afterTextChanged(s: Editable) {
        afterTextChangedCallback?.invoke(s)
    }
}

private typealias BeforeTextChangedFunction =
  (text: String, start: Int, count: Int, after: Int) -> Unit
private typealias OnTextChangedFunction =
  (text: String, start: Int, before: Int, count: Int) -> Unit
private typealias AfterTextChangedFunction =
  (s: Editable) -> Unit
```

下列代码显示了 MainActivity 的 onCreate 方法中的应用过程。

```
package com.sample.marvelgallery.view.main

import android.os.Bundle
import android.support.v7.widget.GridLayoutManager
import android.view.Window
import com.sample.marvelgallery.R
import com.sample.marvelgallery.data.MarvelRepository
import com.sample.marvelgallery.model.MarvelCharacter
import com.sample.marvelgallery.presenter.MainPresenter
import com.sample.marvelgallery.view.common.BaseActivityWithPresenter
import com.sample.marvelgallery.view.common.addOnTextChangedListener
import com.sample.marvelgallery.view.common.bindToSwipeRefresh
import com.sample.marvelgallery.view.common.toast
import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : BaseActivityWithPresenter(), MainView {

   override var refresh by bindToSwipeRefresh(R.id.swipeRefreshView)
   override val presenter by lazy
     { MainPresenter(this, MarvelRepository.get()) }

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       requestWindowFeature(Window.FEATURE NO TITLE)
       setContentView(R.layout.activity main)
       recyclerView.layoutManager = GridLayoutManager(this, 2)
       swipeRefreshView.setOnRefreshListener { presenter.onRefresh() }
       searchView.addOnTextChangedListener {
           onTextChanged { text,  ,  ,   ->
               presenter.onSearchChanged(text)
           }
       }
       presenter.onViewCreated()
   }

   override fun show(items: List<MarvelCharacter>) {
       val categoryItemAdapters = items.map(::CharacterItemAdapter)
       recyclerView.adapter = MainListAdapter(categoryItemAdapters)
   }

   override fun showError(error: Throwable) {
```

```
        toast("Error: ${error.message}")
        error.printStackTrace()
    }
}
```

至此，人物角色的搜索功能全部结束，读者可构建该应用程序，并以此搜索喜爱的人物角色，如图 9.25 所示。

下面将讨论角色资料的显示问题。



图 9.25

9.1.5　人物角色的资料显示

除了人物角色搜索之外，为了使应用程序功能更加丰富，还应添加人物角色的描述显示功能。也就是说，当用户单击某一人物角色的图像时，须显示其背景资料，其中包括角色的名称、相片、描述及其出现次数。

当实现这一用例时，需要生成新的 Activity 和布局，并定义 Activity 的外观。对此，可在 com.sample. marvelgallery.view.character 包中创建名为 Character ProfileActivity 的新 Activity，如图 9.26 所示。

下面首先从布局变化（位于 activity_character_ profile.xml 中）开始讨论。图 9.27 显示了最终的效果。

图 9.26

图 9.27

其中，基本元素则是包含 AppBar 和 CollapsingToolbarLayout 的 CoordinatorLayout，用以实现素材设计中所指的收缩效果，如图 9.28 所示。



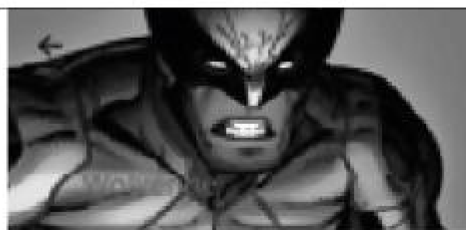




图 9.28

另外，还须针对描述和出现次数定义 TextView，并通过下一个用例中的数据进行填充。activity_character_profile 布局的完整定义如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:id="@+id/character detail layout"
   android:layout width="match parent"
   android:layout height="match parent"
   android:background="@android:color/white">

   <android.support.design.widget.AppBarLayout
       android:id="@+id/appBarLayout"
       android:layout width="match parent"
       android:layout height="wrap content"
       android:theme="@style/ThemeOverlay.AppCompat.ActionBar">

       <android.support.design.widget.CollapsingToolbarLayout
           android:id="@+id/toolbarLayout"
           android:layout width="match parent"
           android:layout height="match parent"
           app:contentScrim="?attr/colorPrimary"
           app:expandedTitleTextAppearance="@style/ItemTitleName"
           app:layout scrollFlags="scroll|exitUntilCollapsed">
```

```
            <android.support.v7.widget.AppCompatImageView
                android:id="@+id/headerView"
                android:layout_width="match_parent"
                android:layout_height="@dimen/character_header_height"
                android:background="@color/colorPrimaryDark"
                app:layout_collapseMode="parallax" />

            <android.support.v7.widget.Toolbar
                android:id="@+id/toolbar"
                android:layout_width="match_parent"
                android:layout_height="?attr/actionBarSize"
                android:background="@android:color/transparent"
                app:layout_collapseMode="pin"
                app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />

    </android.support.design.widget.CollapsingToolbarLayout>
</android.support.design.widget.AppBarLayout>

<android.support.v4.widget.NestedScrollView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:overScrollMode="never"
    app:layout_behavior="@string/appbar_scrolling_view_behavior">

    <LinearLayout
        android:id="@+id/details_content_frame"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:focusableInTouchMode="true"
        android:orientation="vertical">

        <TextView
            android:id="@+id/descriptionView"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:gravity="center"
            android:padding="@dimen/character_description_padding"
            android:textSize="@dimen/standard_text_size"
            tools:text="This is some long text that will be visible as
an character description." />

        <TextView
            android:id="@+id/occurrencesView"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
```

```
            android:padding="@dimen/character description padding"
            android:textSize="@dimen/standard text size"
            tools:text="He was in following comics:\n* KOKOKO \n* KOKOKO
\n* KOKOKO \n* KOKOKO \n* KOKOKO \n* KOKOKO \n* KOKOKO \n* KOKOKO \n*
KOKOKO \n* KOKOKO \n* KOKOKO " />
       </LinearLayout>

   </android.support.v4.widget.NestedScrollView>

   <TextView
       android:layout width="match parent"
       android:layout height="wrap content"
       android:layout gravity="bottom"
       android:background="@android:color/white"
       android:gravity="bottom|center"
       android:text="@string/marvel copyright notice" />

   <ProgressBar
       android:id="@+id/progressView"
       style="?android:attr/progressBarStyleLarge"
       android:layout width="wrap content"
       android:layout height="wrap content"
       android:layout gravity="center"
       android:visibility="gone" />

</android.support.design.widget.CoordinatorLayout>
```

同时，还需要向 styles.xml 中添加下列样式：

```
<resources>

   <!-- Base application theme. -->
   <style name="AppTheme"
          parent="Theme.AppCompat.Light.DarkActionBar">
       <!-- Customize your theme here. -->
       <item name="colorPrimary">@color/colorPrimary</item>
       <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
       <item name="colorAccent">@color/colorAccent</item>
   </style>
   <style name="AppFullScreenTheme"
          parent="Theme.AppCompat.Light.NoActionBar">
       <item name="android:windowNoTitle">true</item>
       <item name="android:windowActionBar">false</item>
       <item name="android:windowFullscreen">true</item>
       <item name="android:windowContentOverlay">@null</item>
```

```
   </style>

   <style name="ItemTitleName"
          parent="TextAppearance.AppCompat.Headline">
       <item name="android:textColor">@android:color/white</item>
       <item name="android:shadowColor">@color/colorPrimaryDark</item>
       <item name="android:shadowRadius">3.0</item>
   </style>
   <style name="ItemDetailTitle"
          parent="@style/TextAppearance.AppCompat.Small">
       <item name="android:textColor">@color/colorAccent</item>
   </style>

</resources>
```

此外，针对 AndroidManifest 中的 CharacterProfileActivity，还须定义 AppFullScreen Theme 作为主题，如下所示：

```
<activity android:name=".view.CharacterProfileActivity"
   android:theme="@style/AppFullScreenTheme" />
```

图 9.29 显示了定义完毕后的布局效果。

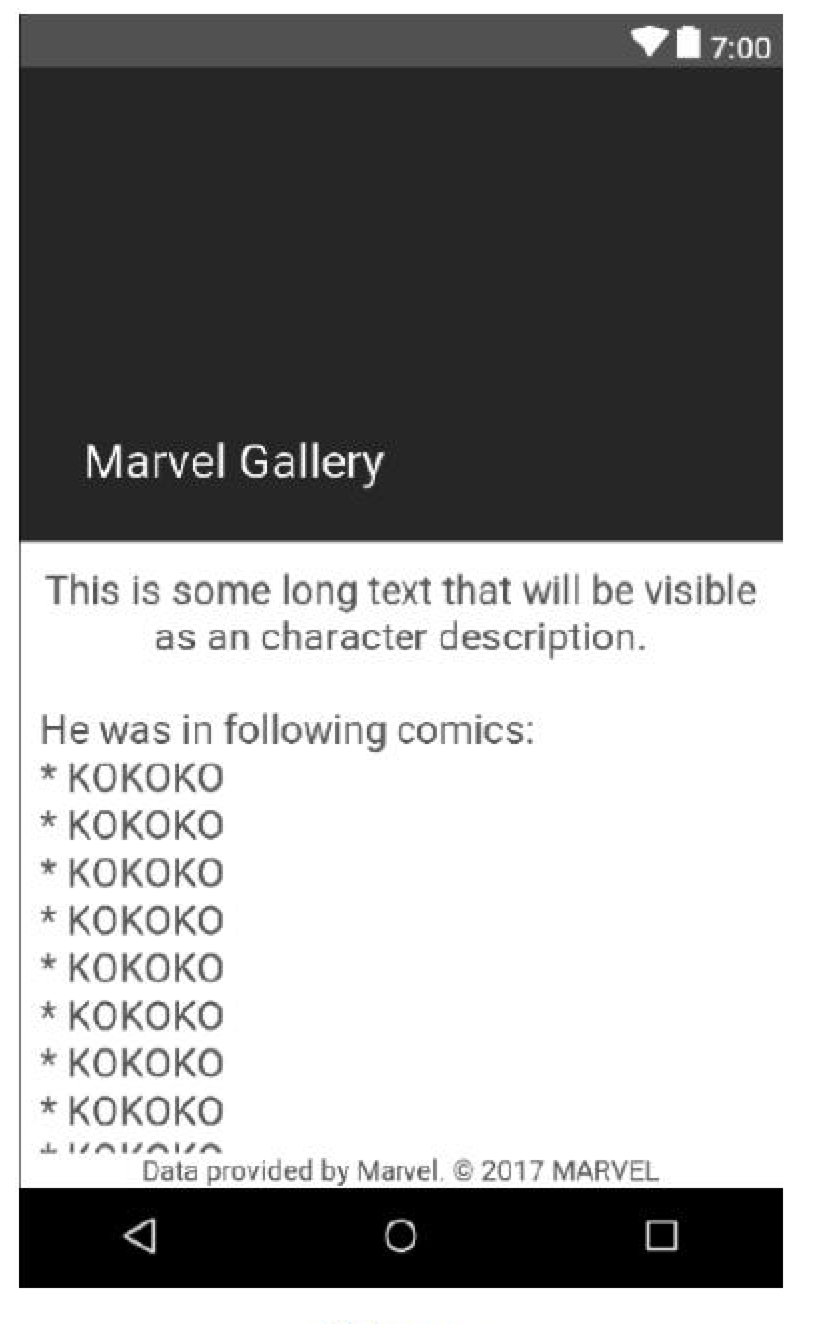


图 9.29

该视图用于显示与人物角色相关的数据，首先需要从 MainActivity 中打开，设置 CharacterItemAdapter 中的 onClickListener，这将调用构造函数提供的 clicked 回调，如下所示：

```
package com.sample.marvelgallery.view.main

import android.support.v7.widget.RecyclerView
import android.view.View
import android.widget.ImageView
import android.widget.TextView
import com.sample.marvelgallery.R
import com.sample.marvelgallery.model.MarvelCharacter
import com.sample.marvelgallery.view.common.ItemAdapter
import com.sample.marvelgallery.view.common.bindView
import com.sample.marvelgallery.view.common.loadImage

class CharacterItemAdapter(
       val character: MarvelCharacter,
       val clicked: (MarvelCharacter) -> Unit
) : ItemAdapter<CharacterItemAdapter.ViewHolder>(R.layout.item character) {

   override fun onCreateViewHolder(itemView: View) =
   ViewHolder(itemView)

   override fun ViewHolder.onBindViewHolder() {
       textView.text = character.name
       imageView.loadImage(character.imageUrl)
       itemView.setOnClickListener { clicked(character) }
}

   class ViewHolder(itemView: View) :
   RecyclerView.ViewHolder(itemView) {
       val textView by bindView<TextView>(R.id.textView)
       val imageView by bindView<ImageView>(R.id.imageView)
   }
}
```

同时还需要更新 MainActivity，如下所示：

```
package com.sample.marvelgallery.view.main

import android.os.Bundle
import android.support.v7.widget.GridLayoutManager
import android.view.Window
import com.sample.marvelgallery.R
```

```
import com.sample.marvelgallery.data.MarvelRepository
import com.sample.marvelgallery.model.MarvelCharacter
import com.sample.marvelgallery.presenter.MainPresenter
import com.sample.marvelgallery.view.character.CharacterProfileActivity
import com.sample.marvelgallery.view.common.BaseActivityWithPresenter
import com.sample.marvelgallery.view.common.addOnTextChangedListener
import com.sample.marvelgallery.view.common.bindToSwipeRefresh
import com.sample.marvelgallery.view.common.toast
import kotlinx.android.synthetic.main.activity main.*

class MainActivity : BaseActivityWithPresenter(), MainView {

   override var refresh by bindToSwipeRefresh(R.id.swipeRefreshView)
   override val presenter by lazy
   { MainPresenter(this, MarvelRepository.get()) }

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       requestWindowFeature(Window.FEATURE NO TITLE)
       setContentView(R.layout.activity main)
       recyclerView.layoutManager = GridLayoutManager(this, 2)
       swipeRefreshView.setOnRefreshListener { presenter.onRefresh() }
       searchView.addOnTextChangedListener {
           onTextChanged { text, , , ->
               presenter.onSearchChanged(text)
           }
       }
       presenter.onViewCreated()
   }

   override fun show(items: List<MarvelCharacter>) {
       val categoryItemAdapters =
       items.map(this::createCategoryItemAdapter)
       recyclerView.adapter = MainListAdapter(categoryItemAdapters)
   }

   override fun showError(error: Throwable) {
       toast("Error: ${error.message}")
       error.printStackTrace()
   }

   private fun createCategoryItemAdapter(character: MarvelCharacter)
           = CharacterItemAdapter(character,
```

```
            { showHeroProfile(character) })

    private fun showHeroProfile(character: MarvelCharacter) {
        CharacterProfileActivity.start(this, character)
    }
}
```

在上述实现中，使用了源自 CharacterProfileActivity 伴生对象中的方法，以启动 CharacterProfileActivity。对此，需要向该方法传递 MarvelCharacter 对象。相应地，传递 MarvelCharacter 对象最为高效的方式是以打包方式对其进行传递。针对于此，MarvelCharacter 须实现 Parcelable 接口。这也是一些有效方案会使用某些注解处理库的原因，例如 Parceler、PaperParcel 或 Smuggler，进而生成所需的元素。此处将使用项目中现有的、源自 Kotlin Android 扩展的解决方案。在本书编写时，该方案仍处于试验阶段，因而需要在 build.gradle 模块中添加下列定义：

```
androidExtensions {
    experimental = true
}
```

全部工作是在类前添加 Parcelize 注解，且需要该类实现 Parcelable。除此之外，为了隐藏默认的 Android 警告消息，还须禁止错误输出，如下所示：

```
package com.sample.marvelgallery.model

import android.annotation.SuppressLint
import android.os.Parcelable
import com.sample.marvelgallery.data.network.dto.CharacterMarvelDto

import kotlinx.android.parcel.Parcelize
@SuppressLint("ParcelCreator")
@Parcelize

    constructor(dto: CharacterMarvelDto) : this(
            name = dto.name,
            imageUrl = dto.imageUrl
    )
}
```

至此，可实现 start 函数以及 character 字段，这将通过属性委托从 Intent 处获取参数，如下所示：

```
package com.sample.marvelgallery.view.character

import android.content.Context
```

```
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.view.MenuItem
import com.sample.marvelgallery.R
import com.sample.marvelgallery.model.MarvelCharacter
import com.sample.marvelgallery.view.common.extra
import com.sample.marvelgallery.view.common.getIntent
import com.sample.marvelgallery.view.common.loadImage
import kotlinx.android.synthetic.main.activity character profile.*

class CharacterProfileActivity : AppCompatActivity() {

   val character: MarvelCharacter by extra(CHARACTER ARG) // 1

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContentView(R.layout.activity character profile)
       setUpToolbar()
       supportActionBar?.title = character.name
       headerView.loadImage(character.imageUrl, centerCropped = true) // 1
   }

   override fun onOptionsItemSelected(item: MenuItem): Boolean = when {
       item.itemId == android.R.id.home -> onBackPressed().let { true }
       else -> super.onOptionsItemSelected(item)
}
   private fun setUpToolbar() {
       setSupportActionBar(toolbar)
       supportActionBar?.setDisplayHomeAsUpEnabled(true)
   }

   companion object {

       private const val CHARACTER ARG =
"com.sample.marvelgallery.view.character.CharacterProfileActivity.Chara
cter ArgKey"

      fun start(context: Context, character: MarvelCharacter) {
          val intent = context
                  .getIntent<CharacterProfileActivity>() // 1
                  .apply { putExtra(CHARACTER ARG, character) }
          context.startActivity(intent)
      }
   }
}
```

对于注释 1，前述内容已对 extra 和 getIntent 扩展函数有所讨论，但尚未在当前项目中予以实现。另外，loadImage 将会显示一条错误消息。

此处需要更新 loadImage，并将 extra 和 getIntent 定义为顶级函数，如下所示：

```
// ViewExt.kt
package com.sample.marvelgallery.view.common

import android.app.Activity
import android.content.Context
import android.content.Intent
import android.os.Parcelable
import android.support.annotation.IdRes
import android.support.v4.widget.SwipeRefreshLayout
import android.widget.ImageView
import android.widget.Toast
import com.bumptech.glide.Glide
import kotlin.properties.ReadWriteProperty
import kotlin.reflect.KProperty
import android.support.v7.widget.RecyclerView
import android.view.View

fun <T : View> RecyclerView.ViewHolder.bindView(viewId: Int)
      = lazy { itemView.findViewById<T>(viewId) }

fun ImageView.loadImage(photoUrl: String, centerCropped: Boolean = false)
{
   Glide.with(context)
           .load(photoUrl)
           .apply { if (centerCropped) centerCrop() }
           .into(this)
}

fun <T : Parcelable> Activity.extra(key: String, default: T? = null):
Lazy<T>
      = lazy { intent?.extras?.getParcelable<T>(key) ?: default ?: throw
Error("No value $key in extras") }

inline fun <reified T : Activity> Context.getIntent() = Intent(this,
T::class.java)
// ...
```



如果不打算定义函数以启动 Activity，则可使用生成此类方法的库。例如，可使用 ActivityStarter 库。下列代码显示了 CharacterProfileActivity。

```
class CharacterProfileActivity : AppCompatActivity() {

   @get:Arg val character: MarvelCharacter by argExtra()

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContentView(R.layout.activity character profile)
       setUpToolbar()
       supportActionBar?.title = character.name
       headerView.loadImage(character.imageUrl, centerCropped = true) // 1
}

   override fun onOptionsItemSelected(item: MenuItem): Boolean = when {
       item.itemId == android.R.id.home -> onBackPressed().let { true }
       else -> super.onOptionsItemSelected(item)
   }

   private fun setUpToolbar() {
       setSupportActionBar(toolbar)
       supportActionBar?.setDisplayHomeAsUpEnabled(true)
   }
}
```

随后，可通过类 CharacterProfileActivityStarter 的静态方法获取其 Intent，进而予以启用，如下所示：

```
CharacterProfileActivityStarter.start(context, character)
val intent = CharacterProfileActivityStarter.getIntent(context, character)
```

对此，需要在 build.gradle 模块使用到 kapt 插件（用以支持 Kotlin 中的注解处理），如下所示：

```
apply plugin: 'kotlin-kapt'
```

build.gradle 模块中的 ActivityStarter 依赖项如下所示：

```
implementation
'com.github.marcinmoskala.activitystarter:activitystarter:1.00'
implementation 'com.github.marcinmoskala.activitystarter: activitystar
terkotlin:1.00'
kapt 'com.github.marcinmoskala.activitystarter:activitystartercompiler:1.00'
```

经过上述调整后，当单击 MainActivity 中的角色人物时，将启用 CharacterProfile Activity，如图 9.30 所示。

当前操作显示了名称以及人物角色的照片。下一步是显示描述内容和出现次数，对应数据位于 Marvel API 中，仅需要扩展 DTO 模块即可获取此类数据。对此，需要添加用于加载列表的 ListWrapper，如下所示：

```
package com.sample.marvelgallery.data.network.dto

class ListWrapper<T> {
    var items: List<T> = listOf()
}
```

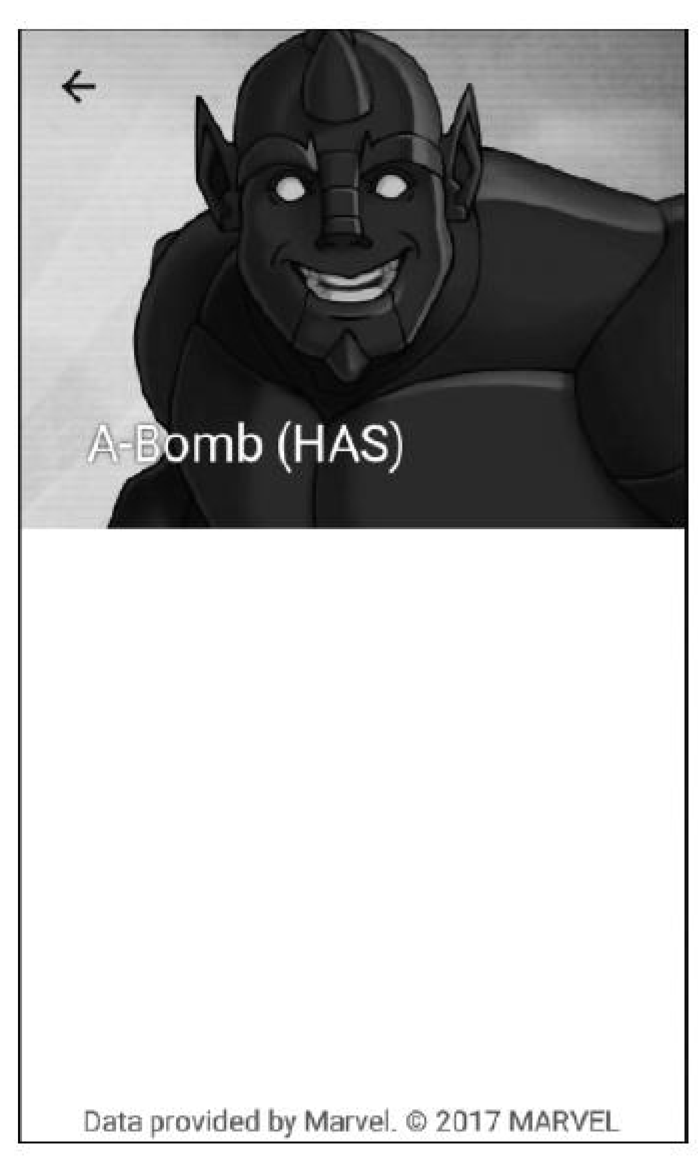


图 9.30

同时，还须定义 ComicDto，用于加载与出现次数相关的数据，如下所示：

```
package com.sample.marvelgallery.data.network.dto

class ComicDto {
    lateinit var name: String
}
```

下面将更新 CharacterMarvelDto，如下所示：

```
package com.sample.marvelgallery.data.network.dto

class CharacterMarvelDto {
```

```
    lateinit var name: String
    lateinit var description: String
    lateinit var thumbnail: ImageDto
    var comics: ListWrapper<ComicDto> = ListWrapper()
    var series: ListWrapper<ComicDto> = ListWrapper()
    var stories: ListWrapper<ComicDto> = ListWrapper()
    var events: ListWrapper<ComicDto> = ListWrapper()

    val imageUrl: String
        get() = thumbnail.completeImagePath
}
```

当前，数据将从 API 中读取，并保存至 DTO 中，但当在项目中对其加以使用时，还须进一步调整 MarvelCharacter 类定义，并添加新的构造函数，如下所示：

```
@SuppressLint("ParcelCreator")
@Parcelize

class MarvelCharacter(
        val name: String,
        val imageUrl: String,
        val description: String,
        val comics: List<String>,
        val series: List<String>,
        val stories: List<String>,
        val events: List<String>
) : Parcelable {

    constructor(dto: CharacterMarvelDto) : this(
            name = dto.name,
            imageUrl = dto.imageUrl,
            description = dto.description,
            comics = dto.comics.items.map { it.name },
            series = dto.series.items.map { it.name },
            stories = dto.stories.items.map { it.name },
            events = dto.events.items.map { it.name }
    )
}
```

现在更新 CharacterProfileActivity，并显示描述内容和出现次数列表，如下所示：

```
class CharacterProfileActivity : AppCompatActivity() {

   val character: MarvelCharacter by extra(CHARACTER ARG)
   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContentView(R.layout.activity character profile)
       setUpToolbar()
       supportActionBar?.title = character.name
       descriptionView.text = character.description
       occurrencesView.text = makeOccurrencesText() // 1
       headerView.loadImage(character.imageUrl, centerCropped = true)
   }

   override fun onOptionsItemSelected(item: MenuItem): Boolean = when {
       item.itemId == android.R.id.home -> onBackPressed().let { true }
       else -> super.onOptionsItemSelected(item)
   }

   private fun setUpToolbar() {
       setSupportActionBar(toolbar)
       supportActionBar?.setDisplayHomeAsUpEnabled(true)
   }

   private fun makeOccurrencesText(): String = "" // 1, 2
           .addList(R.string.occurrences comics list introduction,
character.comics)
           .addList(R.string.occurrences series list introduction,
character.series)
           .addList(R.string.occurrences stories list introduction,
character.stories)
           .addList(R.string.occurrences events list introduction,
character.events)

   private  fun  String.addList(introductionTextId:  Int,  list:  List<
String>):String { // 3
       if (list.isEmpty()) return this
       val introductionText = getString(introductionTextId)
       val listText = list.joinToString(transform =
           { " $bullet $it" }, separator = "\n")
       return this + "$introductionText\n$listText\n\n"
   }
```

```
   companion object {
       private const val bullet = '\u2022' // 4
       private const val CHARACTER ARG =
"com.naxtlevelofandroiddevelopment.marvelgallery.presentation.heroprofi
le.CharacterArgKey"

       fun start(context: Context, character: MarvelCharacter) {
           val intent = context
                   .getIntent<CharacterProfileActivity>()
                   .apply { putExtra(CHARACTER ARG, character) }
           context.startActivity(intent)
       }
   }
}
```

对于注释 1，出现次数列表的构成相对复杂，因而可将其从 makeOccurrencesText 函数中提取出来。此处，针对每个出现种类（漫画、系列等），仅当存在对应类型的出现次数时，方需要显示介绍文本。另外，还需要利用项目符号标注各个数据项。对于注释 2，makeOccurrencesText 表示为独立表达式函数，并使用 addList，通过下一个需要显示的列表附加于（初始状态下）空字符串。如果所提供的列表为空，那么返回的字符串将不会产生任何变化；否则将会返回一个字符串，该字符串添加了介绍文本以及包含项目符号的元素列表。对于注释 4，表示为用作项目符号的字符。

除此之外，还须在 strings. xml 中定义字符串，如下所示：

```
<resources>
   <string name="app_name">Marvel Gallery</string>
   <string name="marvel copyright notice">
       Data provided by Marvel. . 2017 MARVEL</string>
   <string name="search hint">Search for character</string>
   <string name="occurrences_comics_list_introduction">Comics:</string>
   <string name="occurrences series list introduction">Series:</string>
   <string name="occurrences stories list introduction">Stories:</string>
   <string name="occurrences events list introduction">Events:</string>
</resources>
```

图 9.31 显示了人物角色的全部资料，包括人物角色名称、图像、描述内容，以及漫画、系列、事件和故事中的出现次数列表。



图 9.31

9.2 本章小结

本章所介绍的小型项目仍存在许多需要完善的地方。在该应用程序中，讨论了一些 Kotlin 的应用示例，并以此简化 Android 开发。当然，读者还可进一步研究其他处理方案。Kotlin 大大简化了 Android 的开发任务，例如监听器设置、视图元素引用等诸多常见操作，同时还包括一些高级功能，例如过程式编程和集合处理操作。

本书无法兼顾 Kotlin Android 开发中的全部内容，仅介绍了重要的概念和特性。下一步即是打开 Android Studio，编写自己的项目，并享受 Kotlin 的开发之旅，冒险历程已呈现在您的眼前！